Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Tesis de Maestría en Ingeniería en Computación

Especificación de un marco de pruebas asociado a Genexus© con adaptación de funcionalidades de FIT[1].

Alejandro Araújo Pérez

Directoras: Ing. Ana Asuaga
Msc. Ing. Beatriz Pérez Lamacha

Montevideo, Uruguay
Octubre de 2008

---

[1] FIT (*Framework for Integrated Tests*) : Autor Cunningham W.
© Marca registrada de Artech Consultores SRL

Especificación de un marco de pruebas asociado a GeneXus con adaptación de funcionalidades de FIT.
Alejandro Araújo Pérez.
Tesis de Maestría en Ingeniería en Computación
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, Octubre de 2008.

A mis hijos, Mathías y Santiago.
A mis padres, Glady y Roberto.

AGRADECIMIENTOS

Agradezco en general a todas aquellas personas que de una u otra manera estuvieron relacionadas con el proceso de este trabajo de tesis.
A Ana Asuaga y Beatriz Pérez, por su permanente apoyo, comprensión, dedicación, minuciosa y crítica lectura de este trabajo. El permanente entusiasmo de Beatriz y su gestión vinculándome y abriéndome numerosas puertas ha sido de inestimable valor.
A Nicolás Jodal por brindarme su certera guía en la elección de la temática, por su dedicación y disponibilidad brindadas. Los recursos facilitados por él y su organización fueron de vital importancia para la concreción del proyecto.
A Gustavo Proto, por su dedicación, disponibilidad, aportes, la minuciosa y reiterada lectura de la propuesta y su validación. A José Lamas y su equipo de Desarrollo de Artech, por el tiempo dispensado y el soporte técnico brindado durante el desarrollo de las versiones de la herramienta GXUnit.
A Andrés Aguiar, por su dedicación y aportes en las etapas iniciales de este trabajo.
A Enrique Almeida, proponente y mentor de GXUnit, por su apoyo, ideas, difusión de la propuesta y su imprescindible participación para la concreción de las primeras versiones de la herramienta.
A Uruguay Larre Borges, incansable y crítico lector que revisó una y otra vez esta tesis, por el tiempo dedicado, sus sugerencias, su permanente y contagioso optimismo y su activa colaboración en el proyecto GXUnit.
A los estudiantes de los dos grupos de Proyecto de Ingeniería de Software del año 2007 por haber desarrollado la herramienta GXUnit: Adrián García, Anthony Figueroa, Antonio Malaquina, Cecilia Apa, Darío de León, Diego Gawenda, Diego San Esteban, Federico Parins, Fernando Colman, Ken Tenzer, Lucía Adinolfi, Marcelo Falcón, Rafael Sisto, Rodrigo Ordeix, Fernando Varesi, Gervasio Marchand, Guillermo Pérez, Guillermo Polito, Horacio López, Ignacio Esmite, Marcelo Celio, Marcelo Vignolo, Martín Sellanes, Nicolás Alvarez de Ron, Rodrigo Aguerre, Rosana Robaina, Soledad Pérez, Stephanie De León.
A Jorge Triñañes y a todo el equipo docente de Proyecto de Ingeniería de Software 2007, por haber aceptado y apoyado el proyecto.
A Maria Urquhart, por su gestión para viabilizar la propuesta.
A Fernando Brum, por sus brillantes clases sobre Ingeniería de Software, por el apoyo y guía brindada en las etapas previas a este proceso.
A Lorenzo Balerio, Pablo Balerio y Gustavo Balerio, sin su especial apoyo y comprensión no me hubiera sido posible transitar este camino.
A mis compañeros y a los docentes de los diferentes cursos de la Maestría
Finalmente, a mi familia y mis seres queridos, por haber soportado estoicamente mis ausencias y por haberme apoyado en forma incondicional y permanente.

RESUMEN

El *software* presenta una vertiginosa tendencia hacia la ubicuidad. En la medida que los sistemas informáticos adquieren un rol cada vez más relevante se ha vuelto imprescindible que posean la máxima calidad, así como la habilidad de cumplir los requerimientos que se les hubieran establecido. Para obtener confianza en que el *software* trabajará como es debido se lo debe evaluar. Esta evaluación, especialmente dificultosa debido a que el *software* es discreto, intangible, invisible e interdependiente, se ha convertido en una disciplina crítica dentro de la Ingeniería de *Software* y es parte esencial de cualquier proceso de desarrollo; debiendo realizarse con la mayor eficacia (para detectar el mayor número de problemas) y con la mayor eficiencia (para disminuir su costo).

Una parte importante del *software* creado en nuestro medio se desarrolla con GeneXus, herramienta de especificación de sistemas de información que permite generar y mantener bases de datos y código, apoyada en una metodología incremental e iterativa. GeneXus captura los requerimientos a partir de las visiones de los usuarios, modelando la realidad mediante un conjunto de instancias de "objetos tipos" que almacena en una base de conocimiento. Provee los mecanismos de síntesis e inferencia capaces de construir y mantener bases de datos óptimas y los programas de aplicación, minimizando el impacto de los cambios. Sin embargo, en el área de pruebas, GeneXus no brinda actualmente funcionalidades específicas.

El *Framework for Integrated Tests* (FIT), por otra parte, propone una metodología para mejorar la comunicación de los requerimientos, mediante ejemplos representados de forma tabular que simultáneamente pueden utilizarse para probar el *software*. Sin embargo, sus ventajas se diluyen rápidamente al crecer la cantidad de requerimientos y ante el impacto de los cambios.

En este trabajo se realiza un estudio del estado del arte en lo referente a las pruebas, procesos de pruebas en las metodologías ágiles, automatización de las pruebas unitarias (XUnit), FIT y GeneXus, con el fin de elaborar la especificación inicial de un marco de pruebas unitarias asociado a GeneXus para probar los programas producidos por instancias de sus objetos tipo, utilizando funcionalidades inspiradas en aquellas ofrecidas por FIT.

El marco para pruebas que se propone se apoya en GeneXus para asegurar la portabilidad de los casos de prueba, minimizar el impacto de los cambios y crear el código necesario para ejecutar las pruebas en el lenguaje de destino. Brinda un marco de definición y ejecución de pruebas unitarias automatizadas, en un ámbito donde no existe actualmente tal posibilidad y otorga un mecanismo que posibilita la aplicación práctica de funcionalidades de FIT.

Palabras Claves: Automatización de pruebas de programas, FIT, *Framework for Integrated Tests*, GeneXus, GXFIT, GXUnit, Prueba de aceptación, Prueba unitaria.

TABLA DE CONTENIDO

CAPÍTULO 1 .... **23**

PRESENTACIÓN **23**

1.1 MOTIVACIÓN: LAS PRUEBAS DE PROGRAMAS EN AMBIENTE DE DESARROLLO CON GENEXUS ... 24
1.2 DEFINICIÓN DEL PROBLEMA 25
1.3 JUSTIFICACIÓN 26
1.4 OBJETIVO 27
1.5 METODOLOGÍA 28
1.6 CONTRIBUCIÓN 28
1.7 ORGANIZACIÓN DEL DOCUMENTO 29

CAPÍTULO 2 .... **31**

PRUEBAS DE *SOFTWARE* **31**

2.1 DEFINICIONES 32
2.1.1 Introducción 32
2.1.2 Errores, Defectos (Faltas) y Fallas 32
2.1.3 Requerimientos 33
2.1.4 Verificación y Validación 33
2.1.5 Técnicas Estáticas 34
2.1.6 Testing 36
2.1.7 Componentes de la prueba 38
2.1.8 Oráculos 38
2.2 CONSIDERACIONES 39
2.2.1 Alcance de la prueba 39
2.2.2 Aspectos sicológicos 43
2.3 TIPOS DE PRUEBA 43
2.3.1 Prueba estructural y prueba funcional 43
2.3.2 Prueba unitaria, de integración y del sistema 44
2.3.3 Efectuadas por clientes o usuarios 46
2.4 TÉCNICAS PARA LA PRUEBA 47
2.4.1 Técnicas de Caja Negra 47
2.4.2 Técnicas de Caja Blanca 50
2.4.3 Técnicas basadas en la experiencia o en la intuición 50
2.5 PATRONES 52

CAPÍTULO 3 .... **55**

LA VERIFICACIÓN Y VALIDACIÓN EN LAS METODOLOGÍAS ÁGILES **55**

3.1 METODOLOGÍAS ÁGILES 56
3.1.1 Caractestíscas Generales 56
3.2 VERIFICACIÓN Y VALIDACIÓN SEGÚN LAS METODOLOGÍAS ÁGILES 57
3.2.1 *Agile Testing* 57
3.2.2 Aspectos relevantes en cada metodología 59
3.3 *TEST DRIVEN DEVELOPMENT* 66
3.3.1 Características 67
3.3.2 Consideraciones 70
3.3.3 Evolución 71

C A P Í T U L O 4 .... **73**

AUTOMATIZACIÓN DE LAS PRUEBAS **73**

4.1 AUTOMATIZACIÓN 74
4.1.1 Introducción 74
4.1.2 Objetivos 74
4.1.3 Clasificación de las herramientas 75
4.1.4 Estrategias 75
4.1.5 Consideraciones 77
4.2 TECNOLOGÍA XUNIT PARA PRUEBAS UNITARIAS 79
4.2.1 Introducción 79
4.2.2 Escritura de las pruebas 82
4.2.3 Ejecución de las pruebas 86
4.2.4 Terminología 86
4.2.5 Arquitectura 88
4.3 DOBLES PARA PRUEBAS 89
4.3.1 Introducción 89
4.3.2 Clasificación 90
4.3.3 Consideraciones sobre utilización y diseño 93

C A P Í T U L O 5 .... **95**

FIT: "*FRAMEWOK FOR INTEGRATED TESTS*" **95**

5.1 INTRODUCCIÓN 96
5.2 CARACTERÍSTICAS BÁSICAS 96
5.2.1 Tablas y *Fixtures* 98
5.2.2 Secuencia de tablas 106
5.3 FLUJOS DE TRABAJO 107
5.3.1 *DoFixture* 107
5.3.2 *SetUpFixture* 110
5.4 ARQUITECTURA E IMPLEMENTACIÓN 112
5.4.1 Arquitectura 112
5.4.2 Instalación y ejecución 112
5.4.3 Implementaciones 113
5.5 CONSIDERACIONES 113

C A P Í T U L O 6 .... **115**

GENEXUS **115**

6.1 PRESENTACIÓN 116
6.2 CARACTERÍSTICAS 116
6.2.1 Modelo de datos 116
6.2.2 Atributos 118
6.2.3 Variables 118
6.2.4 Dominios 118
6.2.5 Base de conocimiento (KB) y modelos GeneXus 119
6.2.6 Distribución y Consolidación 119
6.2.7 Metodología 119
6.2.8 Aplicaciones y Bases de Datos 121
6.2.9 Ambiente de desarrollo integrado 122
6.2.10 Objetos GeneXus 122
6.3 HERRAMIENTAS ADICIONALES 132
6.4 GENEXUS X 132
6.4.1 Descripción 132
6.4.2 Arquitectura básica 134

6.4.3 *Data Selectors* (DS) 135
6.4.4 *Data Providers* (DP) 136
6.4.5 Ciclo de vida y base de conocimiento 137
6.4.6 Administración de versiones 137
6.5 CONSIDERACIONES 138

C A P Í T U L O 7 .... **139**

PROPUESTA DE ESPECIFICACIÓN **139**

7.1 INTRODUCCIÓN 140
7.1.1 Antecedentes 140
7.1.2 Presentación 141
7.1.3 Objetivos 143
7.1.4 Plataforma tecnológica 144
7.2 HIPÓTESIS Y RESTRICCIONES 144
7.2.1 Supuestos 144
7.2.2 Restricciones 145
7.3 ALCANCE 145
7.3.1 Objetivos específicos 145
7.4 REQUERIMIENTOS 147
7.4.1 Visión 147
7.4.2 Descripción funcional 147
7.4.3 Características técnicas generales 149
7.4.4 Relatos 149
7.5 MODELO DE DOMINIO 163

C A P Í T U L O 8 .... **167**

CONCLUSIONES **167**

8.1 OBJETIVOS ALCANZADOS 168
8.2 APORTES Y LIMITACIONES 169
8.3 INVESTIGACIÓN Y TRABAJO FUTURO 170

BIBLIOGRAFÍA **173**

GLOSARIO **195**

A N E X O A **197**

EL PROYECTO GXUNIT **197**

A.1 RESEÑA 198
A.1.1 Antecedentes 198
A.1.2 Requerimientos 199
A.1.3 Principales características 201
A.2 GXUNIT1 203
A.2.1 Documentación técnica sobre GeneXus 203
A.2.2 Características de la solución técnica 204
A.2.3 Casos de Uso 205
A.2.4 Arquitectura 208
A.2.5 GXUnit1 en acción 211
A.2.6 Grilla del editor de la prueba 216
A.2.7 Comprobación de la base de datos 216
A.3 GXUNIT2 218
A.3.1 Características de la solución técnica 218
A.3.2 Casos de Uso 222

A.3.3 Modelo de Dominio 225
A.3.4 Arquitectura 226
A.3.5 GXUnit2 en acción 230

A N E X O B 239

EXTREME PROGRAMMING (XP) 239

B.1 RESEÑA 240
B.2 DESCRIPCIÓN ORIGINAL DE XP 242
B.2.1 Definición 242
B.2.2 Valores 243
B.2.3 Principios 243
B.2.4 Ciclos de vida 244
B.2.5 Prácticas 245
B.2.6 Roles 247
B.2.7 Círculo de vida y adaptación 247
B.3 EL NUEVO XP (XP2) 248
B.3.1 Valores fundamentales 248
B.3.2 Principios 248
B.3.3 Prácticas 249
B.3.4 Roles 250
B.4 CORRELACIÓN DE PRÁCTICAS XP VS. XP2 251
B.5. CONSIDERACIONES 252

A N E X O C 255

FITNESSE 255

C.1 INTRODUCCIÓN 256
C.2 ARQUITECTURA 260
C.3 INTEGRACIÓN CON OTRAS HERRAMIENTAS 261
C.3.1 FitNesse y Selenium 261
C.3.2 Integración con otras herramientas 263
C.4 CONSIDERACIONES 264

ÍNDICE DE FIGURAS

FIG.2- 1: PATRÓN SIMPLE . 53
FIG.2- 2: PATRÓN SIMPLE CON DATOS . 53
FIG.2- 3: PATRÓN COLECCIÓN 54
FIG.2- 4: PATRÓN PROCESO EN SECUENCIA . 54

FIG. 3- 1: PASOS EN UN CICLO TDD. 68

FIG. 4- 1: MATRIZ DE SELECCIÓN. 77
FIG. 4- 2: CICLO DE VIDA DE TESTCASE . 80
FIG. 4- 3: CODIFICACIÓN EN JUNIT 83
FIG. 4- 4: CODIFICACIÓN EN JUNIT 83
FIG. 4- 5: ESPERANDO UNA EXCEPCIÓN EN JUNIT 84
FIG. 4- 6: *THEORIS* EN JUNIT 84
FIG. 4- 7: *THEORYS* EN JUNIT . 85
FIG. 4- 8: PRUEBAS PARAMETRIZADAS EN JUNIT . 85
FIG. 4- 9: NUNIT. 86
FIG. 4- 10: DIAGRAMA BÁSICO DE CLASES XUNIT . 88
FIG. 4- 11: PATRONES DE DISEÑO APLICADOS A JUNIT . 88
FIG. 4- 12: CÓDIGO Y USO DE UN *STUB* . 90
FIG. 4- 13: CÓDIGO Y USO DE UN *STUB* "*SABOTEUR*" . 91
FIG. 4- 14: CÓDIGO Y USO DE *MOCKS* 92
FIG. 4- 15: CÓDIGO Y USO DE *MOCKS* (2) 93

FIG. 5- 1: FIT: *FIXTURES* Y TABLAS . 97
FIG. 5- 2: ESQUEMA DE TRABAJO CON FIT . 98
FIG. 5- 3: CASO DE PRUEBA UTILIZANDO *COLUMNFIXTURE* Y RESULTADO DE LA PRUEBA. 99
FIG. 5- 4: TABLA DE TIPO *COLUMNFIXTURE* Y FRAGMENTO DE CÓDIGO C# QUE LA INTERPRETA. 101
FIG. 5- 5: CASO DE PRUEBA DE TIPO *ACTIONFIXTURE* Y RESULTADO DE LA PRUEBA. 102
FIG. 5- 6: CASO DE PRUEBA DE TIPO *ACTIONFIXTURE* Y FRAGMENTO DE CÓDIGO C# INTERPRETE. 103
FIG. 5- 7: CASO DE PRUEBA *ROWFIXTURE* Y RESULTADO DE LA PRUEBA. 104
FIG. 5- 8: CASO DE PRUEBA *ROWFIXTURE* INDICANDO COMPROBAR ORDEN . 105
FIG. 5- 9: CÓDIGO DE EJEMPLO PARA IMPLEMENTACIÓN DE *ROWFIXTURE* 105
FIG. 5- 10: SECUENCIA DE TABLAS . 106
FIG. 5- 11: FIT EJECUTANDO 107
FIG. 5- 12: CASO DE PRUEBA DE TIPO *DOFIXTURE* Y RESULTADO DE LA PRUEBA. 108
FIG. 5- 13: INTERPRETACIÓN DE FILAS PARA FORMAR NOMBRES DE MÉTODOS EN *DOFIXTURE*. 108
FIG. 5- 14: EJEMPLO DE IMPLEMENTACIÓN DE *DOFIXTURE* 109
FIG. 5- 15: FIT EJECUTANDO *DOFIXTURE* 110
FIG. 5- 16: FLUJO INCLUYENDO *SETUPFIXTURE* 111
FIG. 5- 17: CÓDIGO C# DE EJEMPLO PARA *SETUPFIXTURE* . 111
FIG. 5- 18: DIAGRAMA DE CLASES ESTÁNDARES DE FIT. 112

FIG. 6- 1: METODOLOGIA: CICLOS DISEÑO-PROTOTIPO Y DISEÑO-PRODUCCIÓN 120
FIG. 6- 2: MODELOS GENEXUS 120
FIG. 6- 3: GENERADORES 121
FIG. 6- 4: MÚLTIPLES BASES DE DATOS. 121
FIG. 6- 5: GENEXUS DEVELOPMENT ENVIRONMENT 122
FIG. 6- 6: CREACIÓN DE OBJETO 123
FIG. 6- 7: DEFINICIÓN DE UNA TRANSACCIÓN 123
FIG. 6- 8: DEFINICIÓN DE UN ATRIBUTO . 124
FIG. 6- 9: FORMULARIO *WINDOWS* DE LA TRANSACCIÓN 124

FIG. 6- 10: FORMULARIO *WEB* D E LA TRANSACCIÓN …. ......……………………………………124
FIG. 6- 11: BUSINESS COMPONENT VARIABLE DE TIPO BC 127
FIG. 6- 12: VARIABLE DE TIPO BC 127
FIG. 6- 13: CODIFICACIÓN DE ACCIONES CON UN BC 127
FIG. 6- 14: ESTRUCTURA DE UN SDT 128
FIG. 6- 15: XML A PARTIR DE SDT...129
FIG. 6- 16: CREACIÓN DE OBJETO REPORTE. ………….. 130
FIG. 6- 17: REPORTE A PARTIR DE UNA TRANSACCIÓN. 130
FIG. 6- 18: DISPOSICIÓN PRELIMINAR DEL INFORME..130
FIG. 6- 19: CÓDIGO GENERADO AUTOMÁTICAMENTE. 130
FIG. 6- 20: EJEMPLO DE USO DE COMANDO CALL...131
FIG. 6- 21: IDE VERSIÓN X -WORKFLOW- 134
FIG. 6- 22: IDE CON EXTENSIONES 134
FIG. 6- 23. ARQUITECTURA BÁSICA 134
FIG. 6- 24: ARQUITECTURA Y EXTENSIONES 134
FIG. 6- 25: ESTRUCTURA DE UN *DATA SELECTOR* 135
FIG. 6- 26: SINTAXIS Y EJEMPLO DE *DATA SELECTOR* EN FOR EACH 136

FIG. 7- 1: OBJETOS PARA PRUEBA. 143
FIG. 7- 2: MODELO DE DOMINIO GENERAL DE GXUNIT-GXFIT..164
FIG. 7- 3: MODELO DE OBJETOS VERIFICADORES GXUNIT-GXFIT. 165
FIG. 7- 4: MODELO DE TABLAS DE PARÁMETROS GXUNIT-GXFIT. 165
FIG. 7- 5: MODELO DE UN CASO DE PRUEBA GXUNIT-GXFIT. 166
FIG. 7- 6: MODELO DE DOBLES PARA PRUEBAS GXUNIT-GXFIT. 166

FIG. A- 1: ARQUITECTURA DE GENEXUS ROCHA (GXUNIT1)..203
FIG. A- 2: ESTRUCTURA DE CLASES SEGÚN LA INVESTIGACIÓN PARA GXUNIT1. 204
FIG. A- 3: DIAGRAMA DE CASOS DE USO (GXUNIT1)..206
FIG. A- 4: CASOS DE USO RELEVANTES A LA ARQUITECTURA (GXUNIT1)..........................208
FIG. A- 5: DESCOMPOSICIÓN EN SUBSISTEMAS (GXUNIT1) 209
FIG. A- 6: MODELO DE DISTRIBUCIÓN (GXUNIT1). 211
FIG. A- 7: REGLA "*PARM*"DE PROCEDIMIENTO A PROBAR (GXUNIT1). 211
FIG. A- 8: "*SOURCE*"DE PROCEDIMIENTO A PROBAR (GXUNIT1). 212
FIG. A- 9: EDITOR DE PRUEBAS GXUNIT1. 212
FIG. A- 10: CREACIÓN DEL PROCEDIMIENTO VERIFICADOR (GXUNIT1). 213
FIG. A- 11: "*BUILD*" (GXUNIT1). 213
FIG. A- 12: MENÚ PARA EJECUTAR PRUEBAS (GXUNIT1)..213
FIG. A- 13: SELECCIÓN DE PRUEBAS A EJECUTAR (GXUNIT1). 214
FIG. A- 14: VISTA DE LOS RESULTADOS DE LAS PRUEBAS -1ER..NIVEL- (GXUNIT1)...........................214
FIG. A- 15: VISTA DE LOS RESULTADOS DE LAS PRUEBAS –EXPANDIDA- (GXUNIT1). 214
FIG. A- 16: VISTA DE LOS RESULTADOS DE LAS PRUEBAS –EXPANDIDA- (GXUNIT1). 216
FIG. A- 17: GRILLA DEL EDITOR DE LAS PRUEBAS (GXUNIT1). 216
FIG. A- 18: OBJETO TESTSET (GXUNIT2)...219
FIG. A- 19: CASO DE PRUEBA (GXUNIT2)...219
FIG. A- 20: OBJETO PVU (GXUNIT2)..219
FIG. A- 21: CONCEPTOS (GXUNIT2)...220
FIG. A- 22: DIAGRAMA DE CASOS DE USO (GXUNIT2)...223
FIG. A- 23: MODELO DE DOMINIO (GXUNIT2). 225
FIG. A- 24: CASOS DE USO RELEVANTES A LA ARQUITECTURA (GXUNIT2)........................227
FIG. A- 25: DESCOMPOSICIÓN EN SUBSISTEMAS (GXUNIT2). 227
FIG. A- 26: SUBSISTEMA GXUNIT (GXUNIT2). 227
FIG. A- 27: INTERFASE IWEBSERVICE (GXUNIT2)..228
FIG. A- 28: DIAGRAMA DE CLASES (GXUNIT2). 229
FIG. A- 29: CREACIÓN DE OBJETO TESTSET (GXUNIT2)...230
FIG. A- 30: EDITOR DEL OBJETO TESTSET (GXUNIT2). 230

Fig. A- 31: Procedimiento verificable a probar y sus parámetros (GXUnit2). 231
Fig. A- 32: Editor de TestSet y su grilla para ingreso de casos de prueba (GXUnit2)....... 231
Fig. A- 33: Combo de PVUs en el editor del TestSet (GXUnit2). 232
Fig. A- 34: Menú para ejecución de pruebas (GXUnit2)... 232
Fig. A- 35: Ventana para lanzar ejecución de pruebas (GXUnit2). 233
Fig. A- 36: Otra vista de la ventana para lanzar pruebas (GXUnit2). 233
Fig. A- 37: Reporte de ejecución (GXUnit2). 234
Fig. A- 38: Menú GXUnit (GXUnit2). 234
Fig. A- 39: Visor de *Logs* (GXUnit2).. 235
Fig. A- 40: Otra vista del visor de *Logs* (GXUnit2). 235
Fig. A- 41: Impacto en el TestSet por cambio en la regla *parm* (GXUnit2). 236
Fig. A- 42: Impacto en el TestSet por cambio en PVUs (GXUnit2). 237
Fig. A- 43: Configuración del nivel de detalle del *log* (GXUnit2). 237
Fig. A- 44: Ruta donde se ubica el *log* (GXUnit2). 237

Fig. B- 1: Raíces de eXtreme Programming 240
Fig. B- 2 (a) Modelo en cascada, (b) Modelo iterativo (ej.Espiral), (c) XP. 241
Fig. B- 3: Costo del cambio. 242
Fig. B- 4: Escalas de tiempo en XP 245
Fig. B- 5: Circulo de vida.. 247

Fig. C- 1: Página inicial de FitNesse. 257
Fig. C- 2: Creación de una nueva página. ……... 257
Fig. C- 3: Página nueva, lista para el ingreso de información. 257
Fig. C- 4: Editando una página para indicar vínculo a páginas de ejemplo. 258
Fig. C- 5: Página pronta para ser usada ………. 258
Fig. C- 6: Página inicial de ejemplos….. 258
Fig. C- 7: Página con conjunto (suite) de casos de prueba. 258
Fig. C- 8: Nueva página con conjunto de casos de prueba. 258
Fig. C- 9: Caso de prueba “Descuentos”……... 258
Fig. C- 10: Edición de un caso de prueba. 259
Fig. C- 11: Caso de prueba “Descuentos” modificado…... 259
Fig. C- 12: Resultado de ejecutar el caso de prueba “Descuentos”. 259
Fig. C- 13: Arquitectura de FitNesse 260
Fig. C- 14: FitNesse Distribuido. 261
Fig. C- 15: Integración FitNesse - Selenium….. 262
Fig. C- 16: Herramienta Stiq (StoryTestIQ)…. 262
Fig. C- 17: FITClipse 264
Fig. C- 18: Descargas de FitNesse. 265

ÍNDICE DE CUADROS Y TABLAS

TABLA 4- 1: COMPARATIVO DE TERMINOLOGÍA XUNIT 87

TABLA 5- 1: *FIXTURES* BÁSICAS. 98

TABLA 6- 1 EJEMPLOS DE DATA PROVIDERS 137
TABLA 6- 2 EJEMPLOS DE UTILIZACIÓN DE DATA PROVIDERS 137

TABLA A- 1: ARCHIVO XML CON DATOS DE PRUEBA SDT (GXUNIT1). 215

TABLA B- 2: CORRELACIÓN DE PRÁCTICAS ENTRE XP Y XP2. 251

C a p í t u l o 1

PRESENTACIÓN

En este capítulo se expone la motivación, se define el problema, se indican los objetivos, se explica la metodología utilizada y se enuncian las principales contribuciones de esta tesis. Finalmente se brinda una guía para el lector acerca de la organización del documento.

El presente capítulo está organizado de la siguiente manera:

- En la sección 1.1 se presenta la motivación de este trabajo: Las pruebas de programas en ambientes de desarrollo con GeneXus.
- En la sección 1.2 se define el problema.
- En la sección 1.3 se ofrece una justificación acerca de la elección de las herramientas en las que se apoya la propuesta.
- En la sección 1.4 se explicitan los objetivos.
- En la sección 1.5 se describe la metodología utilizada.
- En la sección 1.6 se enuncian las principales contribuciones de esta tesis.
- En la sección 1.7 se explica la organización del resto del documento.

1.1 Motivación: Las pruebas de programas en ambiente de desarrollo con GeneXus

A lo largo de veintiocho años de experiencia en informática he participado en numerosos proyectos de desarrollo de *software*, durante los cuales cometí errores y aprendí lecciones, habiendo buscado respuestas a una serie de interrogantes:

- ¿Estaré produciendo el producto que el cliente requirió?
- ¿Qué puedo hacer para entender mejor sus requerimientos?
- ¿Juzgará al producto como de buena calidad?
- ¿El producto tendrá muchas fallas?
- ¿Cómo evitar mayores desvíos en el tiempo y costo previsto?
- ¿Cómo mitigar el impacto ante los cambios tecnológicos?
- ¿Se podrá adaptar el producto frente a una realidad cambiante sin introducir demasiados defectos?

Con el objetivo de mejorar en las dimensiones de tiempo, costo y calidad, en la administración del cambio, independencia tecnológica y captura más adecuada de los requerimientos, dadas las dificultades inherentes a la característica esencial de invisibilidad del *software*, me lancé a la búsqueda de una "bala de plata" [Bro95] que me llevó, en cambio, a descubrir la herramienta para desarrollo GeneXus [GX07] (ver capítulo sexto)[2] producida por la empresa uruguaya Artech. Convencido que me permitiría mejorar en varios de los aspectos mencionados, adherí a su propuesta, con lo cual logré efectivamente disminuir radicalmente la cantidad de tareas accidentales y el tiempo empleado en su ejecución, enfocándome en resolver las esenciales.

Mejorar en las dimensiones mencionadas e independizarme en cierto grado del cambio tecnológico me permitió desarrollar, utilizando GeneXus, aplicaciones más grandes y capaces de resolver requerimientos más complejos, las cuales hubo que probar sin la ayuda específica de GeneXus, más allá de la facilidad dada por la prototipación local inmediata, la cual hace un aporte sustancial a la validación de los requerimientos con el usuario con la consiguiente disminución de probables errores en su captura, pero no establece por si misma una nueva dirección en pruebas [RBG98].

La experiencia adquirida en el área me ha permitido comprobar la aseveración acerca que las pruebas de *software* insumen una porción importante del tiempo en los proyectos [Bro95] [Mye04] [Bei90]; o referida al costo, sobre lo cual según cita Tassey [Tas02] puede elevarse hasta un 75% del total. Más preocupante aún, mi experiencia me ha permitido comprobar cuán fácil es ingresar en el círculo vicioso de más presión por terminar, menos pruebas, más errores [Bec02].

[2] De ahora en adelante las referencias a capítulos, secciones y apartados de esta tesis se harán colocando su número entre paréntesis curvos.

Dando por cierto que cuanto más temprana sea la detección de los errores será menor el impacto de los mismos y el costo de su corrección; coincido en que una infraestructura para pruebas efectiva y que acerque la corrección de los errores al momento de su inyección en el ciclo de vida disminuirá dramáticamente los costos [Tas02]. Introducir *software* que permita automatizar tareas de ingeniería y operaciones referidas a las pruebas (automatización de las pruebas), conjugado con las herramientas de desarrollo, es uno de los caminos a transitar.

Surge entonces el desafío consistente en trabajar en la elaboración de un mecanismo que permita probar más eficaz y eficientemente programas elaborados con GeneXus suministrando al mismo tiempo un vehículo complementario para la mejor comprensión de los requerimientos, mediante ejemplos.

1.2 Definición del problema

GeneXus permite crear y administrar modelos (de ahora en adelante bases de conocimiento) a partir de los cuales genera y mantiene automáticamente tanto bases de datos como código de la aplicación en diferentes lenguajes y plataformas; acompañando el ciclo de vida y brindando portabilidad a nuevas tecnologías, minimizando los cambios a realizar para lograrlo. Potencia la capacidad de entregas tempranas y prototipado, mediante un desarrollo incremental e iterativo (IID), coincidente con recomendaciones metodológicas de amplia aceptación y buenas prácticas reconocidas en la industria. Sin embargo, en el área de pruebas, no brinda actualmente funcionalidades específicas que se aproximen en versatilidad y potencia a las requeridas. El desarrollador se ve beneficiado por la rapidez de desarrollo y generación, un incremento sustancial en su productividad, el alto nivel de abstracción que le permite la herramienta, la creación y mantenimiento totalmente automático de las bases de datos en función de las visiones de los usuarios y las enormes ventajas dadas por el prototipado inmediato y local, pero no tiene ayudas específicas para la automatización de las pruebas.

Por otra parte el *Framework for Integrated Tests* (FIT) (5) [Cun207] propone un mecanismo para mejorar la comunicación y colaboración durante el proceso de desarrollo de *software*, al tiempo que permite la automatización de las pruebas, guiado por comandos y datos, pero carece de un marco apropiado para su instrumentación. Existen en el mercado numerosas implementaciones para FIT, entre las cuales se destaca FitNesse (Anexo C) [MMW07], que suscitó un importante entusiasmo inicial el cual ha decaído debido a los inconvenientes que surgen en la práctica, cuando aumenta la cantidad y la complejidad de los casos de prueba, especialmente en los aspectos de reestructuración de los mismos ante cambios en el código o los requerimientos.

El problema que se aborda en este trabajo puede resumirse en las siguientes interrogantes:

- ¿Es posible crear un marco para automatizar las pruebas unitarias de objetos GeneXus?
- ¿Qué requerimientos debería cumplir dicho marco para aprovechar las características de GeneXus en la elaboración de programas para prueba y el mantenimiento de los casos de prueba en forma automatizada?
- ¿Cómo adaptar funcionalidades inspiradas en FIT para dicho marco de pruebas unitarias de programas producidos por GeneXus teniendo en cuenta sus particularidades, posibilidades y características?

La propuesta contenida en esta tesis y el trabajo relacionado a la misma en pro de la elaboración de una herramienta de pruebas para GeneXus tienen como objetivo aportar respuestas a las preguntas planteadas.

1.3 Justificación

Se elige GeneXus basándose en las siguientes consideraciones:

- Es una herramienta de especificación de sistemas de información basada en la aplicación de un modelo matemático que permite integrar las visiones de los usuarios en bases de conocimiento a partir de las cuales genera, mediante ingeniería inversa y procesos de inferencia, bases de datos óptimas y aplicaciones completas; pudiendo mantenerlas en forma automatizada ante cambios en los requerimientos [Art105] [LSLN03] [Sal06].
- Trabaja sobre especificaciones, lo cual permite independencia tecnológica.
- Permite acceder y modificar las bases de conocimiento mediante programas no desarrollados por la empresa productora. En particular su nueva versión ofrecerá la posibilidad de programar extensiones al producto [GCW107].
- Una parte considerable del total de *software* producido en Uruguay se elabora con esta herramienta creada y mantenida por una empresa uruguaya, Artech, de penetración a nivel mundial.
- Se cuenta con experiencia personal de 11 años en contacto con GeneXus.
- En el área de pruebas dicha herramienta no brinda actualmente una funcionalidad específica que se aproxime en versatilidad y potencia al resto de sus características.
- En la comunidad de sus usuarios se han lanzado varias propuestas para la construcción de herramientas que automaticen tareas de validación y verificación. A modo de ejemplo pueden citarse el proyecto “GXUnit” [Alm04] [Gxu106] [Gxu108] [Gxu208] [Gxu07] el cual se vincula con este trabajo, y la propuesta de generación automática de casos de prueba [WV05].

Se escoge utilizar conceptos aportados por FIT (5) basándose en las siguientes consideraciones:

- Utiliza un formato tabular como soporte para los casos de prueba. Se cree en que dicho formato resulta inteligible al usuario y al desarrollador; siendo un

vehículo apropiado tanto para expresar los casos de prueba como para explicitar requerimientos mediante ejemplos. El valor de este tipo de representaciones ha sido reconocido y utilizado desde hace largo tiempo. [JPZ96].
- Permite expresar en forma no ambigua y simple los resultados de las pruebas.
- Puede utilizarse tanto para pruebas de aceptación, para las que fue diseñado, como para pruebas unitarias.
- Se conduce con comandos y datos, lo cual brinda ductilidad e importantes ventajas frente a otro tipo de técnicas de automatización [FG99].
- Su utilización requiere estrechar el vínculo entre desarrolladores, usuarios y verificadores (*testers*).
- Despertó gran interés, especialmente entre lo practicantes de las metodologías ágiles [MA01]; si bien actualmente su aplicación práctica se ha visto ralentizada y puesta seriamente en duda (5.5). Los problemas que se presentan con el mantenimiento de los casos de prueba son del tipo de aquellos que típicamente resuelve bien GeneXus.
- Los componentes a ser programados se escriben en el mismo lenguaje que el sistema a probar (SUT).

1.4 Objetivo

El objetivo consiste en elaborar un documento conteniendo requerimientos para un marco (*framework*) para pruebas unitarias asociado a GeneXus, que incluya una propuesta de adaptación de funcionalidades del *Framework for Integrated Tests* (FIT), especificándose una implementación que contemple:

- Creación y mantenimiento automatizado de programas especializados en pruebas unitarias parametrizables, de programas producidos a partir de objetos GeneXus.
- Obtención desde tablas de la especificación (parámetros) de los casos de prueba.
- Ejecución de los casos de prueba.
- Registro de los resultados.

Para satisfacer el objetivo se producen los artefactos que se enumerarán a continuación:

- Documento de Requerimientos.
- Modelo de dominio.

Se participa además activamente del proyecto de especificación y elaboración de la herramienta GXUnit, como prueba de concepto.

1.5 Metodología

La metodología de trabajo ha comprendido las siguientes actividades:

- Investigar acerca de GeneXus, FIT y xUnit (6) (5) (4.2).
- Abordar el estado del arte de las pruebas de *software* a través de libros, artículos, ponencias, manuales, cursos y seminarios, abarcando enfoques tradicionales y tendencias actuales, especialmente en lo referido a pruebas unitarias y de aceptación.
- La participación activa en el proyecto para la creación de una herramienta para pruebas unitarias inspirada en los marcos xUnit: "El Proyecto GXUnit".

En el marco del proceso de elaboración de este trabajo se establecieron vínculos con Artech, el Centro de Ensayos de Software[3], la Cátedra de Ingeniería de Software de la Facultad de Ingeniería de la Universidad de la República, el Centro de Posgrados y Actualización Profesional de la Facultad de Ingeniería de la Universidad de la República y la comunidad de desarrolladores con GeneXus.

1.6 Contribución

Las principales contribuciones de este trabajo se agrupan en tres áreas estrechamente vinculadas:

- El primer aporte consiste en un estudio del estado del arte, en lo que refiere a las pruebas de programas en general y pruebas unitarias y de aceptación en particular. Se estudian definiciones, elementos, técnicas y disciplinas. Se analiza desde el punto de vista de las metodologías ágiles, especialmente desde la perspectiva de *eXtreme Programming* (XP) [BA04] (Anexo B) y se estudia la técnica de desarrollo dirigido por pruebas: "TDD: *Test Driven Development*" [4] (3.3) [Bec02]. Se expone acerca de la automatización de las pruebas en general y se ofrece una descripción de herramientas para pruebas unitarias y de aceptación basadas en código libre, especialmente xUnit y FIT. Se estudia la herramienta de desarrollo GeneXus. Se ofrece una discusión acerca de las disciplinas y herramientas estudiadas. Esta información es organizada y resumida en el presente trabajo para ser utilizada como referencia en la propuesta.
- El segundo aporte surge de la necesidad de impulsar la especificación y construcción de la herramienta para pruebas unitarias en GeneXus sobre la cual aplicar las adaptaciones propuestas. Implica exponer los resultados de la investigación y trabajo realizado en la especificación de requerimientos para responder las dos primeras preguntas: ¿Es posible crear un marco para automatizar las pruebas unitarias de objetos GeneXus requiriendo mínima

[3] Organización dedicada a las pruebas independiente de productos de *software*. http://ces.com.uy
[4] También mencionada como *Extreme Testing* [Mye04].

programación por parte del desarrollador? y ¿qué requerimientos debería cumplir dicho marco para aprovechar características de GeneXus en la elaboración de programas para prueba y el mantenimiento de los casos de prueba en forma automatizada? A tales efectos se definieron los requerimientos mínimos a cumplir por la herramienta, en forma compartida y acordada con los demás proponentes del "Proyecto GXUnit" mencionado en (1.5), incluyéndose parcialmente algunas de las funcionalidades propuestas en la contribución principal de este trabajo, de forma de comprobar su viabilidad. Se propone el proyecto para su realización en el curso "Proyecto de Ingeniería de Software 2007" y una vez aceptado se actúa en el rol de Cliente realizándose el seguimiento de su construcción y validación de los prototipos, junto al resto de los proponentes. El desarrollo de la herramienta en su primera versión ha sido efectuado por dos grupos de estudiantes en el contexto de dicho curso (A) [Gxu07].

- El tercer aporte, la contribución principal de este trabajo, consiste en una propuesta que define y especifica una adaptación de funcionalidades del *Framework for Integrated Tests* aplicable a la prueba unitaria de objetos GeneXus de forma de responder a la pregunta: ¿cómo adaptar funcionalidades inspiradas en FIT para un marco de pruebas unitarias de programas producidos por GeneXus teniendo en cuenta sus particularidades, posibilidades y características? La propuesta se expone explicitándose sus objetivos generales y específicos, los requerimientos comunes y no comunes con GXUnit, conjuntamente con un modelo de dominio y especificación funcional.

1.7 Organización del documento

Esta sección contiene una descripción de la organización del documento, el cual se presenta dividido en ocho capítulos. Se indica un breve resumen del contenido de cada capítulo.

En el capítulo 2 se brinda una visión de las pruebas de *software* en general, definiéndose los términos principales, conceptos, distintos tipos de pruebas y técnicas.

En el capítulo 3 se presentan la verificación y validación desde la perspectiva de las metodologías ágiles y se analiza la disciplina "*Test Driven Development*".

En el capítulo 4 se examinan conceptos generales relativos a la automatización de las pruebas, se estudia la familia de marcos para pruebas unitarias conocida como xUnit y la aplicación de técnicas de reemplazo temporal de unidades de las cuales depende la unidad a probar, en su relación con TDD.

En el capítulo 5 se describe y estudia acerca de FIT, el marco de pruebas de aceptación cuyas funcionalidades inspiran la propuesta contenida en este trabajo.

En el capítulo 6 se describen características generales de GeneXus, detallándose aquellas relacionadas directamente con la propuesta contenida en este trabajo.

En el capítulo 7 se explicita la propuesta (GXFIT) y su relación con GXUnit, el marco de pruebas unitarias asociado a GeneXus: se ofrece el documento de requerimientos y el modelo de dominio.

En el capítulo 8 se extraen conclusiones y se analizan objetivos alcanzados proponiéndose trabajo futuro.

C a p í t u l o   2

PRUEBAS DE *SOFTWARE*

En este capítulo se brinda una visión de las Pruebas de *Software* en general, definiéndose los términos principales, brindándose conceptos, principios, distintos tipos de pruebas, técnicas y patrones.

Está organizado de la siguiente manera:

- En la sección 2.1 se brindan definiciones, conceptos y principios.
- En la sección 2.2 se exponen consideraciones sobre el problema de la limitación de la prueba y aspectos sicológicos de la prueba.
- En la sección 2.3 se presentan tipos de prueba.
- En la sección 2.4 se describen técnicas de prueba.
- En la sección 2.5 se resume sobre patrones para las pruebas.

Se desea destacar que las secciones 2.1 a 2.4 de este capítulo se construyeron utilizando como guía para su elaboración los aportes vertidos en el capítulo segundo y cuarto de la tesis “Proceso de *Testing* Funcional Independiente” de Beatriz Pérez [Per06] y en el capítulo segundo, sección tercera., de la tesis “Mejora de la Calidad de los Prototipos Desarrollados en un Contexto Académico” de Diego Vallespir [Val06].

2.1 Definiciones

2.1.1 Introducción

El *software* presenta una vertiginosa tendencia hacia la ubicuidad. Habiendo abandonado su confinamiento en computadores pasó a imbricarse en una miríada de dispositivos de uso diario.

En la medida que los sistemas informáticos adquieren un rol cada vez más relevante se ha vuelto imprescindible que posean la máxima calidad. Dicha calidad está referida a atributos tales como la robustez, la fiabilidad y la facilidad de uso así como la habilidad de cumplir los requerimientos que se hubieran establecido para los sistemas [Bur03]. Para obtener confianza en que el *software* trabajará como es debido, en el ambiente para el que fue previsto, se lo debe evaluar [Bei90]. Esta evaluación, especialmente dificultosa debido a que el *software* es discreto, intangible, invisible e interdependiente, se ha convertido en una disciplina crítica dentro de la Ingeniería de *Software* y es parte esencial de cualquier proceso de desarrollo. Dada la limitación mostrada por Dijkstra en 1970 "*La Prueba de Software puede ser usada para mostrar la presencia de defectos, pero nunca su ausencia*" [Dij70] debe realizarse con la mayor eficacia (para detectar el mayor número de problemas) y con la mayor eficiencia (para disminuir su costo). Según Huang [Hua75] "*Dada la ausencia de métodos prácticos que puedan ser utilizados para mostrar que un programa está libre de errores, entonces debe ponerse el mayor esfuerzo en mejorar la habilidad de descubrirlos*".

Dicha evaluación puede realizarse desde enfoques dinámicos y enfoques estáticos:

- Los enfoques dinámicos apuntan a ejecutar los programas para determinar si funcionan según lo esperado.
- Los enfoques estáticos se refieren a la evaluación del *software* sin ejecutarlo, (como ser analizar el código, diagramas, documentos de diseño y de requerimientos).

2.1.2 Errores, Defectos (Faltas) y Fallas

Se adoptarán en el contexto de este trabajo las definiciones expresadas en [IEEE Std 610.12-1990]:

- **Equivocación (*mistake*):** Acción del ser humano que produce un resultado incorrecto.
- **Defecto o Falta (*fault*):** Un paso, proceso o definición de dato incorrecto en un programa de computadora. Puede ser el resultado de una equivocación. (Potencialmente origina una falla).
- **Falla (*failure*):** Resultado incorrecto, manifestación de una falta.

- **Error:** Magnitud por la que el resultado es incorrecto.

Una falla corresponde a la manifestación de un defecto. El defecto es lo que la provoca. "*Un defecto es una visión interna del sistema, desde la óptica de los desarrolladores, una falla es una visión externa, un problema que ve el usuario*" [Per06]. No todos los defectos corresponden a una falla.

En la práctica es común intercambiar estos términos asignándoles el mismo significado. En la bibliografía aparece también reiteradamente el termino ***Bug***, que suele utilizarse para referirse a una falta pero que podría también indicar cualquier problema o limitación en el *software*.

2.1.3 Requerimientos

Los requerimientos son propiedades que deben ser exhibidas con el fin de resolver algún problema del mundo real. Los requerimientos del *software* expresan las necesidades y limitaciones puestas en un producto de *software* para resolver un problema del mundo real [SWE04]. Corresponden a lo que debe hacer el *software* y a las características que debe presentar. Los requerimientos del *software* tienen como característica esencial que deben ser comprobables (mediante experimentos).

Se conoce como "especificación de requerimientos del *software*" a la documentación de los requerimientos del *software,* de forma que pueda someterse a una revisación, evaluación y aprobación, de manera sistemática [Per06].

2.1.4 Verificación y Validación

Los términos Verificación y Validación (V&V) denominan al conjunto de procesos y técnicas de evaluación de *software*. Estos términos suelen ser usados indistintamente, pero en realidad implican conceptos diferentes, según las diferentes definiciones que pueden encontrarse en la bibliografía.

El objetivo principal de la V&V es detectar defectos en los productos del desarrollo de *software*. Los enfoques estáticos detectan los defectos directamente mientras que los enfoques dinámicos lo hacen en forma indirecta, a través de las manifestaciones externas, que estos defectos producen al ejecutarse el código. La remoción de estos defectos[5] deberá hacerse para lograr un producto de mejor calidad [Val06].

Boehm [Boe84] usa dos preguntas para manifestar la diferencia entre Verificación y Validación:

- **Verificación:** ¿Estamos elaborando correctamente el producto?
- **Validación:** ¿Estamos elaborando el producto correcto?

[5] Las técnicas de remoción de defectos no se tratarán en este trabajo por considerarse fuera del área de V&V.

Según la definición aportada en [IEEE Std 610.12-1990]:

- La **Verificación** es el proceso de evaluación de un sistema o componente para determinar si un producto de una determinada fase de desarrollo satisface las condiciones impuestas al inicio de dicha fase.
- La **Validación** es el proceso de evaluación de un sistema o componente durante o el final del proceso de desarrollo para determinar cuando se satisfacen los requerimientos que se hubieran especificado.

Según CMMI [CMMI02] se definen en función de su propósito, especificándose que:

- **El propósito de la Verificación** es asegurar que los productos internos cumplen con su especificación de requerimientos.
- **El propósito de la Validación** es demostrar que un producto o componente de un producto cumple su uso previsto cuando es puesto en el ambiente para el cual se diseñó.

Las actividades de V&V son fundamentales en todo proceso de desarrollo de *software*.

2.1.5 Técnicas Estáticas

Estas técnicas se basan en la revisión sistemática de artefactos tales como el código y los documentos con las especificaciones y el diseño. Son técnicas realizadas en general por humanos, aunque es posible también utilizar autómatas para realizar ciertos análisis estáticos del código. Myers [Mye04] engloba estas técnicas bajo la denominación "*human testing*". Tienen la particularidad que detectan directamente los defectos.

Las técnicas estáticas son muy efectivas para la prevención de defectos. Ha sido señalado en reiteradas fuentes en la literatura e Internet que estas técnicas se constituyen en la principal fuente de detección de errores. Myers [Mye04] indica que desde un 30% hasta un 70% de todos los defectos encontrados al final de la V&V pueden ser detectados con estas técnicas. Otros autores aseguran que se llega al 90% [Gla02]. Notablemente, el costo por defecto encontrado es más barato que con la aplicación de las técnicas dinámicas. Esto no implica sustituir o eliminar a las técnicas dinámicas, las cuales resultan imprescindibles para la validación.

Las inspecciones y recorridas son las dos técnicas más importantes de tipo estático realizables por personas [Bei90]. Se puede inspeccionar el código, los documentos de requerimientos, u otros, procediendo de forma similar más allá del objeto a verificar.

El objetivo es comprobar la correspondencia entre cierta porción de código y su especificación funcional. Las técnicas estáticas se utilizan para la verificación [Val06].

A continuación se describirán brevemente algunas técnicas. Un análisis detallado se entiende que queda fuera del alcance de este trabajo.

.

2.1.5.1 Inspecciones

Las inspecciones son exámenes visuales de un producto de *software* para detectar anomalías. Involucran necesariamente al autor del producto a revisar e incluyen un líder (facilitador o moderador) entrenado en técnicas de inspección. No deben participar los jefes de los participantes [SW04].

Durante una sesión de inspección ocurren dos actividades principales:

- **El autor expone,** leyéndole a los demás el contenido de un documento (leyendo línea por línea si se trata del código de un programa) mientras los otros intervienen con preguntas y diversas apreciaciones. El simple hecho de leer a una audiencia tiene una remarcable efectividad en la detección de errores. El moderador es responsable de asegurar que la discusión procede por vía productiva y enfocada en la detección de errores, no en su corrección.
- **Se controla contra una lista (histórica) de errores comunes.**

El tiempo óptimo para estas sesiones es de 90 a 120 minutos. Para que sean efectivas se debe establecer una actitud apropiada, donde el autor no debe sentir la inspección como un ataque a su trabajo [Mye04]. Además de la detección de errores tiene efectos colaterales también beneficiosos, en el sentido de la información que puede recibir el autor acerca de técnicas de programación, de diseño o de estilo.

Cuando la inspección la realiza el propio desarrollador sobre su trabajo se conoce como “auto inspección” o “revisión de escritorio”. No es tan efectiva como las inspecciones grupales. Para el caso del código, no implica tomar el lugar de la computadora, sino analizar el código [Bei90].

2.1.5.2 Recorridas (*Walk-throughs*)

Los objetivos de esta técnica se fijan en encontrar anomalías en el producto, mejorarlo, considerar soluciones alternativas y evaluar la conformidad con estándares y normativas. Es similar a una inspección aunque conducida más informalmente. La organiza el líder del equipo para dar la oportunidad de revisar en forma conjunta el trabajo [SW04].

Las recorrida es dirigida por un diseñador o programador y los participantes (miembros del equipo de desarrollo y otros interesados) intervienen haciendo preguntas, formulando críticas y esgrimiendo comentarios sobre probables defectos, errores, violaciones de estándares y otros.

2.1.5.3 Revisiones entre pares

Por "revisiones entre pares" se entiende el proceso de someter el trabajo de una persona al escrutinio de otro experto en la misma área, para que le revise y formule las apreciaciones que entienda conveniente. Aplicado al *software* puede lograrse, por ejemplo, intercambiando entre dos programadores su código para una revisión; o, en una escala de tiempo más inmediata y dinámica, trabajando en parejas.

2.1.6 *Testing*

Para el termino "*Testing*" (referido al *software*) la acepción que parece ser la adoptada en general en la bibliografía se refiere al enfoque dinámico.

La *Guide to the Software Engineering Body of Knowledge* (SWEBOK) [SWE04] define a la **Prueba de *Software* (*Software Testing*)** como:

"*La verificación dinámica del comportamiento de un programa contra el comportamiento esperado, usando un conjunto finito de casos de prueba, seleccionados de manera adecuada desde el dominio infinito de ejecución.*"

Esta definición menciona varios aspectos que resulta importante destacar:

- ***La verificación dinámica***: implica que para realizarla hay que ejecutar el programa para los datos de entrada.
- ***El comportamiento esperado***: debe ser posible decidir cuando la salida observada de la ejecución del programa es aceptable o no. El comportamiento observado puede ser revisado contra las expectativas del usuario, contra una especificación o contra el comportamiento anticipado por requerimientos implícitos o expectativas razonables.
- ***Un conjunto finito de casos de prueba:*** La prueba se deberá efectuar sobre un conjunto, finito y acotado, de los posibles casos de prueba. Un caso de prueba (*test case*) es un conjunto de valores de entrada, resultados esperados pre y poscondiciones de ejecución, desarrollados con un objetivo particular [IEEE Std 610.12-1990].
- ***Seleccionados adecuadamente:*** La selección de los casos de prueba debe maximizar la efectividad de la prueba.
- ***El dominio infinito de ejecución:*** Se parte de la base que los requerimientos de prueba no tienen límite.

En la SWEBOK se declara que el enfoque estático no forma parte propiamente dicha del "*testing*", aunque lo complementa, estudiándose bajo el área de la calidad (*KA Software Quality*), si bien la definición dada para "*testing*" es genérica:

"***Testing*** *es una actividad realizada para evaluar la calidad del producto y mejorarla, identificando defectos y problemas*".

Myers [Mye04] define:

- ***"Testing"*** o "*Computer Testing"***:** "*Es el proceso de ejecutar a un programa con la intención de encontrar errores*"
- ***"Human Testing":*** para referirse al enfoque estático.
- ***"Software Testing"***: "*Es el proceso, o serie de procesos, diseñados para asegurarse que el código hace aquello para lo que fue diseñado y que no hace nada que se supone no deba hacer*".

La guía de principios para la validación de *software* para la industria de la FDA[6], V.2 [FDA-GSV-V2] establece que la "Prueba de *Software*" (*Software Testing*) implica "*ejecutar el software bajo condiciones conocidas con entradas definidas y salidas documentadas que puedan ser comparadas con las expectativas predefinidas*". La define como una de las muchas actividades de verificación que se realizan con el propósito de confirmar que la salida del desarrollo cumple con sus requerimientos de entrada. Entre las demás actividades de verificación incluye a los enfoques estáticos. Advierte, adicionalmente, que en la literatura se han utilizado muchas veces los términos Verificación, Validación y *Testing* como equivalentes.

En [IEEE Std 610.12-1990] se define "*testing"* como:

"*El proceso de operar un sistema o componente bajo condiciones específicas, observando o registrando los resultados, y realizar una evaluación de algún aspecto del dicho sistema o componente*".

Cem Kaner explicita una distinción importante al definir "*testing"* como "*una investigación técnica cuyo objetivo es exponer información relativa a la calidad sobre el producto bajo prueba"*. Su objetivo no debe ser meramente encontrar defectos mediante la provocación de fallas; debe levantarse la mira, permitiendo que las pruebas sirvan como mecanismo de aprendizaje sobre el sistema bajo prueba [Kan04].

En este trabajo se utilizarán los términos **prueba** o **ensayo** para referirse al enfoque dinámico, y en general se referirá a **V&V** cuando se deba abarcar ambos enfoques. Se comparte la apreciación que las actividades de prueba y los artefactos producidos brindan conocimiento sobre el sistema y su comportamiento así como suministran ejemplos que explicitan los requerimientos y pueden guiar el diseño, además de su objetivo primario de detectar fallas. Las métricas asociadas a estas actividades pueden utilizarse no solo como una medida del grado de satisfacción de las pruebas sino también como una medida del avance de los proyectos de desarrollo.

Por último, se desea destacar que otros autores se refieren al "*testing*" de forma más general, incluyendo al enfoque estático. Burnstein [Bur03] le define como "*el conjunto de todas las actividades de V&V*" [Bur03]. McGregor y Sykes [MS01] le

[6] *Food & Drug Administration.*

definen como "*la evaluación de los artefactos creados durante un esfuerzo de desarrollo de software*".

2.1.7 Componentes de la prueba

- Una **prueba** (*test*) es una actividad en la que un sistema o componente es ejecutado bajo condiciones especificadas, los resultados son observados o registrados y una evaluación es hecha de algún aspecto del sistema o componente.
- Un **caso de prueba** (*test case*) es un conjunto de valores de entrada, precondiciones de ejecución, resultados esperados y poscondiciones de ejecución, desarrollados con un objetivo particular o condición de prueba, tal como ejercitar un camino de un programa particular o para verificar que se cumple un requerimiento especifico [IEEE Std 610.12-1990].
- Un **conjunto** (*test suite*) de casos de prueba [IEEE Std 610.12-1990].
- Un **procedimiento de prueba** (*test procedure*) consiste en instrucciones detalladas para la configuración, ejecución y evaluación de los resultados para un caso de prueba determinado.
- Un **resultado real** (*actual result*) es el comportamiento producido u observado cuando un componente o sistema es probado [ISQ07].
- Un **resultado esperado** (*expected result*) es el comportamiento predicho por la especificación u otra fuente, del componente o sistema a ser probado bajo condiciones especificadas [ISQ07].
- Un **ciclo de prueba** (*test cycle*) es la ejecución del proceso de pruebas contra una versión identificada del producto a probar [ISQ07].
- Los **datos de prueba** (*test data)* son los datos existentes antes que una prueba sea ejecutada, y que afectan o son afectados por el componente o sistema a probar [ISQ07].
- La **ejecución de la prueba** (*test execution*) es el proceso de ejecutar una prueba, produciendo el resultado real [ISQ07].
- Un **Guión** (*test script*): es un programa que especifica para la prueba el elemento a ser probado, los requerimientos, el estado inicial, las entradas, los resultados esperados y los criterios de validación [Bei90].

Un caso de prueba puede ser usado en más de un procedimiento de prueba [IEEE Std 610.12-1990].

2.1.8 Oráculos

La prueba implica controlar que se cumplan o no expectativas sobre el comportamiento del sistema y sus resultados. Un oráculo es cualquier agente que decide si un programa pasó una prueba dada, emitiendo un veredicto al respecto (*pass*=pasa, *fail*=falla) [SW04]. Todos los métodos de ejecutar pruebas necesitan un oráculo.

Los oráculos pueden clasificarse como pasivos y activos. Los oráculos pasivos controlan el resultado versus una salida esperada, los activos reproducen (imitan) el comportamiento del programa a prueba. El oráculo más común es el oráculo entrada/salida, que especifica la salida esperada para una entrada específica [Bei90].

Según Binder "*El oráculo perfecto debería ser una implementación totalmente confiable y equivalente a la conducta del SUT. Aceptaría cada entrada del SUT y produciría siempre el resultado correcto…Si un oráculo perfecto fuera posible y estuviera disponible para el ambiente del SUT simplemente se dispensaría al SUT y se utilizaría el oráculo para satisfacer los requerimientos"* [Bin99].

Dado que un programa puede fallar en múltiples maneras, el problema de interpretar los resultados es complejo. Es necesario poder decidir cuándo un programa falla, dado un conjunto de datos de entrada y una salida esperada. Determinar la corrección de los resultados se conoce como el "Problema del Oráculo" y surge cuando no puede definirse el procedimiento de decisión. El tratamiento en detalle de este problema excede el alcance de este trabajo.

2.2 Consideraciones

Se expondrán consideraciones en relación al alcance de la prueba y los aspectos sicológicos de la prueba.

2.2.1 Alcance de la prueba

2.2.1.1 La imposibilidad de la prueba exhaustiva y completa

Por prueba exhaustiva se entiende probar un programa para todas las combinaciones de entradas, en cada punto donde se ingresan datos y bajo cada estado posible del programa, pero lograr una prueba de esta naturaleza es imposible [Bur03] [Mye04] [Kan02].

Huang, con un ejemplo análogo al suministrado por Dijkstra [Dij70] ilustra acerca de la imposibilidad de la prueba completa debido a una explosión combinatoria. Si un programa acepta dos variables de entrada (x e y) y una variable de salida (z), enteras con un valor máximo de 2^{32} , implica que el total posible de combinaciones de valores de entrada es $2^{64.}$ Si se supone que este programa se ejecuta en un milisegundo tomaría millones de años completar la prueba [Hua75]. Cem Kaner menciona que la prueba llevada adelante por Doug Hoffman, probando la función raíz cuadrada entera en el computador *MASPAR,* de 64.000 procesadore*s,* para todos los posibles valores enteros de 32 bits, tomó 6 minutos. Sin embargo, destaca que una prueba análoga involucrando enteros de 64 bits no sería viable, ya que tomaría cerca de 4 millones de horas [Kan00].

Kaner [Kan02] resume acerca de esta imposibilidad indicando las siguientes razones:

- El dominio de las posibles entradas es muy grande.
- Existen excesivas combinaciones de datos a probar.
- La cantidad de caminos posibles dentro del programa a probar es enorme.
- Existen errores de interfase del usuario, de configuración, de compatibilidad y docenas de otras dimensiones de análisis.

El objetivo de la prueba debe ser detectar errores. Probar que un programa está libre de errores es imposible. Manna y Waldinger [MW78] distinguieron tres barreras teóricas acerca de la posibilidad de lograr una prueba completa:

- No es posible estar seguros que las especificaciones son correctas.
- Ningún sistema puede verificar a cada programa correcto.
- No es posible asegurar que un sistema de verificación es correcto.

Por otra parte, desde el punto de vista de la estructura del código, la prueba exhaustiva consistiría en suministrar los casos de prueba necesarios para recorrer todos los caminos posibles y probar todas las combinaciones de decisión posibles. Myers [Mye04] advierte y ejemplifica acerca de la imposibilidad de esta tarea, incluso para programas triviales con un número muy bajo de decisiones.

Se necesitan por lo tanto estrategias para escoger los casos de prueba significativos, definiéndose como tales aquellos que tienen alta probabilidad de provocar una falla. La meta no es ejecutar una gran cantidad de casos de prueba, sino ejecutar un número suficiente de casos de prueba significativos [Val06].

Para efectuar la selección de casos significativos existen diversas técnicas. Una técnica habitual es seleccionar elementos, desde el dominio de entrada, agrupándoles en clases de equivalencia, de las cuales se eligen sus representantes para intervenir en la prueba. También es recomendable efectuar un análisis de riesgo (2.2.1.2) a los efectos de determinar donde poner el énfasis en las pruebas. Inspeccionar el código, para considerar valores de entrada que maximicen la cantidad de líneas ejecutadas y caminos recorridos dentro del código, es otra alternativa a tener en cuenta a la hora de dividir el dominio de entradas. Este tipo de técnicas pueden agruparse bajo la denominación genérica de "*Domain Testing*".

Por último, el énfasis está puesto en la seducción, más que en la deducción. Se debe proveer suficientes pruebas (*enough testing*) para asegurar que la probabilidad de falla es baja. El vocablo "suficiente" implica buen juicio. Lo que es suficiente para probar un juego no lo será para un programa de control de un reactor [Bei90].

2.2.1.2 Determinación del alcance

Existen diversas estrategias que pueden utilizarse para determinar el alcance de las pruebas, entre las cuales se encuentran:

<u>*Good Enough (suficientemente buena)*</u>

James Bach resume su posicionamiento frente al problema de la imposibilidad de la prueba exhaustiva con la aproximación denominada "*Good Enough testing* ", que define de la siguiente manera:

"*Good Enough testing es el proceso de desarrollar una evaluación suficiente sobre la calidad, a un costo razonable, para permitir decisiones sabias y oportunas concernientes al producto*". "*El ensayo perfecto* (*perfect testing*) *es aquel que permite a un involucrado tomar una decisión correcta sobre un producto*" [Bac98].

Este proceso se deriva de un marco más general, también propuesto por Bach, conocido como "*Good Enough Quality*" (GEQ) [Bac97] [Bac98].

<u>*Risk Based (basado en riesgos)*</u>

Un riesgo se define como la probabilidad que algo no previsto ocurra. James Bach [Bac98] enunció que "*cuanto mayor la probabilidad que ocurra un problema, y cuanto mayor el impacto de su ocurrencia, es mayor el riesgo asociado. Entonces, la prueba es motivada por riesgos*". La magnitud del riesgo es proporcional a la probabilidad de ocurrencia del problema y a su impacto.

Bach identifica tres pasos para las pruebas basadas en riesgos:

- Confeccionar una lista priorizada de riesgos.
- Realizar pruebas que exploren cada riesgo.
- Cuando un riesgo se mitiga y emergen otros nuevos, se debe ajustar el esfuerzo de la prueba.

Bach clasifica los métodos para el análisis de riesgos, en dos categorías: heurísticos o rigurosos.

- <u>Los enfoques heurísticos</u> se dividen en dos vertientes complementarias, siendo posible manifestar la diferencia entre ellas mediante las siguientes preguntas:
    - **Desde adentro hacia afuera:** ¿qué riesgos se asocian a esta funcionalidad?
    - **Desde afuera hacia adentro:** ¿qué funcionalidades se asocian a esta clase de riesgo?
- <u>Los enfoques rigurosos</u> utilizan modelos estadísticos o analizan exhaustivamente los riesgos y modos de falla.

2.2.1.3 ¿Cuándo deben detenerse las pruebas?

Myers [Mye04] enumera los siguientes cinco criterios que pueden utilizarse para detener el esfuerzo de pruebas:

- **El tiempo asignado para el mismo expiró**, lo cual es un criterio inútil y muy peligroso, ya que el haber terminado el tiempo asignado no es una medición en absoluto ni de la calidad del producto ni de las pruebas.
- **Se ejecutaron todos los casos de prueba sin detectar fallas,** nuevamente se está frente a otro criterio inútil, una mala calidad de los casos de prueba puede determinar su ejecución sin provocar fallas. El fenómeno también puede darse cuando los casos de prueba se corren una y otra vez, luego de ir levantando los errores, hasta que no detectan fallas, caso que Brian Marick, citado por Cem Kaner en [Kan021], ejemplifica con la parábola de transitar por un mismo sendero, ya libre de minas, en un campo minado: no se tiene ningún dato acerca que el resto del campo esté repleto o no de ellas.
- **Se alcanzó un cierto nivel de cubrimiento.** Si el cubrimiento es referido al código, entonces puede ser a nivel de las pruebas que desarrollan los programadores para las piezas de código o módulos (pruebas unitarias, sobre las cuales se tratará en 2.3.2). El cubrimiento también puede estar referido, a alto nivel, a la funcionalidad cubierta por las pruebas, los requerimientos satisfechos y la satisfacción del cliente con relación a la funcionalidad obtenida y el nivel y características de los fallos conocidos remanentes.
- **La cantidad de defectos remanentes es menor que un número dado**. Este criterio necesita cuantificar la cantidad de defectos remanentes, lo cual solo puede hacerse en base a estimaciones. Se fija una cantidad mínima de defectos remanentes, o un porcentaje del total de defectos estimados, como valor límite para detener las pruebas. En [Val06] se ofrece un resumen acerca de dos de los métodos de estimación: la siembra de defectos y las pruebas independientes.
- **La efectividad de las pruebas ha disminuido hasta un valor predefinido**. Este criterio considera que deben detenerse las pruebas cuando la probabilidad de encontrar nuevos defectos es muy baja para un lapso dado.

El análisis de riesgos viene en ayuda de la determinación de la finalización de las pruebas en la medida que se va obteniendo confianza a medida que se van mitigando los riesgos más prioritarios.

Criterios de Fiabilidad

Existen otros criterios, basados en modelos de fiabilidad, que pueden utilizarse para tomar la decisión de detener las pruebas. Estos modelos se basan en estimar el estado actual y predecir la confiabilidad en un estado futuro. Intentan reflejar un patrón de defectos partiendo del análisis de datos históricos sobre las pruebas. Suelen ser denominados “modelos de la fiabilidad del *software*” (SRM). Una vertiente propone modelos para medir el aumento de la confiabilidad y determinar si el costo de lograr ese crecimiento se justifica.

Existe una gran cantidad de modelos. Entre ellos se encuentran el modelo de Goel-Okumoto y el modelo básico de Musa. Tanto el estudio de la confiabilidad como de los modelos de fiabilidad excede el alcance de esta tesis.

2.2.2 Aspectos sicológicos

Myers [Mye04] menciona varios aspectos sicológicos a tener en cuenta. Entre ellos se describirán los siguientes:

- **La prueba debe ser vista como un proceso destructivo** de tratar de encontrar los errores (cuya presencia es asumida) en un programa. El objetivo que debe fijarse el verificador (*tester*) es detectar fallas; de esa forma, al fijar la meta apropiada, se obtiene un importante efecto sicológico positivo y pone su mayor empeño en detectar fallas. Si hiciera lo contrario, y se fijara como objetivo el comprobar que el programa funciona correctamente, estaría fallando en su propósito cada vez que detecta una falla. Este mismo proceso destructivo puede también explicar la dificultad de los programadores cuando escriben sus propias pruebas, resulta difícil para alguien cuya tarea es construir (*software*) efectuar una actividad que debe ser vista como un proceso "destructivo".
- **La tarea debe ser factible.** Si se fija una tarea que es vista como imposible o no factible por las personas que deben realizar es altamente probable que fracasen en cumplir el objetivo. Definir la prueba como el proceso de encontrar errores lo convierte en una tarea factible.

Cem Kaner, James Bach y Bret Pettichord [KBP01] resaltan la importancia que el rol del verificador sea visto por los demás integrantes del equipo como alguien que, más allá de destruir, también brinda evidencia. Sostienen que la actividad del verificador es epistemología aplicada y que los ensayos están basados en la sicología cognoscitiva, recomendando estudiar sobre dichos tópicos.

2.3 Tipos de prueba

En esta sección se abordará la definición de los diferentes tipos de prueba y sus diversas clasificaciones, siguiendo criterios utilizados por Whittaker [Whi00], la SWEBOK [Swe04] y Myers [Mye04].

2.3.1 Prueba estructural y prueba funcional

En esta dimensión se clasifican las pruebas según se tenga o no conocimiento del código a probar.

- **Caja Blanca o Estructural:** Los casos de prueba se derivan desde la examinación de la estructura o implementación de los programas o componentes a probar. Dado entonces que para su realización se deben conocer las internas del programa o componente, reciben el nombre de "caja blanca".
- **Caja Negra o Funcional:** Se ve al sistema o componente como una "caja negra" de la cual se tiene su especificación, desconociéndose su

implementación, conociéndose su comportamiento a nivel de las entradas y las salidas. Los casos de prueba se derivan del análisis de las especificaciones, siendo su propósito mostrar discrepancias con la especificación y no demostrar que el programa cumple su especificación [Mye04]. La prueba funcional toma el punto de vista del usuario [Bei90].

2.3.2 Prueba unitaria, de integración y del sistema

La prueba es efectuada a diferentes niveles a lo largo del proceso de desarrollo y mantenimiento. El objeto a probar (blanco de la prueba) varía: desde un simple módulo, un grupo de módulos o el sistema entero. En esta dimensión se clasifican las pruebas según su blanco [SWE04].

- **Prueba unitaria:** Es la prueba realizada sobre pequeñas porciones de código individuales o unidades. Una unidad es la pieza de *software* más pequeña que se puede probar. Normalmente refiere al trabajo de un programador [Bei90]. En el paradigma de orientación a objetos (OOLP) la unidad de trabajo se refiere a una clase, a un método de una clase o a un conjunto pequeño de clases (*class-clusters*) [Bin99]. Se busca con su aplicación el beneficio de la detección temprana de defectos. En este sentido resulta oportuno recordar la frase atribuida a Boris Beizer según se cita en [Jfi06] acerca que "*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*"[7]. En el alcance de este trabajo las pruebas unitarias adquieren capital importancia, por lo cual se volverá reiteradas veces sobre su análisis.
- **Prueba de Integración:** Estas pruebas se realizan una vez que los componentes individuales están prontos y debe realizarse la integración con otros. La integración es el proceso por el cual los componentes son agregados. La prueba de integración es efectuada para mostrar que la combinación de componentes es incorrecta o inconsistente [Bei90]. Su objetivo apunta a encontrar fallas en las interacciones entre los componentes.
  Para combinar los componentes existen varias estrategias. Si el *software* está estructurado de manera jerárquica, donde un módulo invoca a otros de nivel inferior y así sucesivamente hasta llegar a uno que hace el trabajo, se puede usar una estrategia incremental. En la estrategia incremental se efectúan pruebas unitarias sobre cada pieza de *software*. Luego se prueban las combinaciones de piezas mediante pruebas de integración. Tanto esta estrategia como otras guiadas por la arquitectura requieren de programas sustitutos provisorios ("dobles para pruebas"), básicamente *stubs* (sustituyen a los módulos "colaboradores") y *drivers* (sustituyen a los módulos que efectúan la invocación), para reemplazar momentáneamente los reales o para disparar y controlar su ejecución, los que se estudiarán más detenidamente en (4.3).

[7] Los autores hacen la salvedad acerca que Beizer reconoció posteriormente que la falla del Therac 25 fue una excepción a esta regla [Jfi06].

- **Prueba del Sistema:** Estas son las pruebas que se realizan sobre el sistema completo. Abarcan tanto pruebas funcionales como no funcionales del sistema. La mayoría de las faltas deben haber sido identificadas durante las pruebas de unidad e integración. La prueba del sistema generalmente se considera apropiada para probar requerimientos no funcionales del sistema tales como seguridad, desempeño, exactitud y confiabilidad. Las interfases externas, los dispositivos de hardware, o el ambiente de funcionamiento también se evalúan [SWE04]. Las pruebas del sistema, a su vez, pueden clasificarse según diferentes tipos. Una clasificación posible es la propuesta por Myers [Mye04], quien ubica a las pruebas del sistema dentro de un grupo de pruebas de "alto nivel" (*high order*), que también incluye a las pruebas de aceptación (2.3.3.1). Dicha clasificación se resume y expone parcialmente a continuación:

    - Prueba de Volumen: La estrategia consiste en someter el sistema a grandes volúmenes de datos. Su propósito es demostrar que no puede manejar el volumen de datos especificados en sus objetivos.
    - Pruebas de Esfuerzo: La estrategia consiste en someter el sistema a cargas pesadas consistentes en un volumen máximo de datos o actividad durante un lapso corto.
    - Pruebas de "Usabilidad": Su objetivo es encontrar problemas con la facilidad de uso, claridad de presentación, y otros, en su interacción con las personas.
    - Pruebas de Seguridad: Intentan violentar las comprobaciones de seguridad del sistema.
    - Pruebas de Desempeño: Su objetivo es demostrar que el sistema no satisface sus objetivos de desempeño o de eficiencia. Estos objetivos generalmente indican tiempos de repuesta y rendimiento.
    - Pruebas de Almacenamiento: Su propósito es demostrar que ciertos objetivos de almacenamiento, como ser la cantidad de memoria o espacio en disco a utilizar, no se han resuelto.
    - Pruebas de Configuración: Prueban el sistema bajo distintas configuraciones de *hardware* y *software.*
    - Pruebas de Compatibilidad: Se diseñan los casos de prueba para demostrar que los objetivos de la compatibilidad no se han resuelto
    - Pruebas de Conversión: Muchos sistemas sustituyen a otros o surgen como evolución de sistemas anteriores, por lo cual es necesaria una conversión de datos. El objetivo de estas pruebas es demostrar que los procedimientos de conversión no funcionan.
    - Pruebas de Instalación: Se prueban aspectos relacionados con la instalación, especialmente cuando esta es automatizada.
    - Pruebas de Confiabilidad: Su objetivo es probar las declaraciones específicas de confiabilidad establecidas para el sistema.

2.3.3 Efectuadas por clientes o usuarios

Existe un conjunto de pruebas que están a cargo de los clientes o usuarios. En estas pruebas que efectúan los clientes o usuarios pueden colaborar otros roles, pero la responsabilidad de la decisión corresponde a los primeros. Se describirá brevemente acerca de las siguientes:

2.3.3.1 Pruebas de Aceptación

Son las pruebas que realizan los clientes o usuarios para conocer si el sistema funciona de acuerdo a los requerimientos y necesidades que establecieron para el mismo. Pueden o no involucrar a los desarrolladores del sistema y a los verificadores (*testers*). Son de fundamental importancia ya que brindan directamente al cliente elementos para su aprobación del sistema.

2.3.3.2 Pruebas Alfa y Beta

Antes de liberar el *software* se distribuye el mismo a un pequeño grupo, representativo de los usuarios potenciales, para su uso interno (alfa) o externo (beta). Estos usuarios reportan problemas con el producto al equipo de desarrollo.

2.3.4 Otros tipos de pruebas

Se describirá brevemente acerca de las pruebas de regresión y de las pruebas de humo.

2.3.4.1 Pruebas de Regresión

Su objetivo es verificar que no ocurrió una regresión en la calidad del producto luego de un cambio, probando que el cambio ha causado que algo que funcionaba dejo de hacerlo. La estrategia consiste en ejecutar nuevamente las pruebas ya realizadas. Deben permitir obtener confianza en que los cambios realizados no afectaron el *software* de maneras no deseadas.

"*Las pruebas de regresión a nivel de la aplicación dan a los usuarios una chance de conversar concretamente acerca de qué está mal y acerca de sus expectativas. Las pruebas de regresión al menor nivel de la escala permiten brindar al desarrollador una forma de mejorar sus pruebas*" [Bec02].

2.3.4.2 Pruebas de Humo

Una prueba de humo es una versión condensada de un conjunto de pruebas de regresión. Se focaliza en probar, a un nivel alto, la funcionalidad más crítica [Dus03]. Las pruebas de humo buscan inestabilidades grandes o elementos clave defectuosos o faltantes, que harían imposible efectuar las pruebas tal como fueron planificadas,

actuando como filtro previo a la realización de otras pruebas. Su nombre deriva de la prueba inicial que realizan técnicos de *hardware* al encender un dispositivo. [KBP01].

2.4 Técnicas para la prueba

Existen numerosas técnicas para efectuar las pruebas. Los tipos de prueba ya vistos pueden utilizar una o varias de estas técnicas. Se agruparán, según el criterio utilizado en [SWE04], de la siguiente manera:

- Técnicas de caja negra (basadas en la especificación).
- Técnicas de caja blanca (basadas en el código).
- Técnicas apoyadas en la experiencia.

A continuación se brindará una breve descripción de las técnicas.

2.4.1 Técnicas de Caja Negra

Se conoce con la denominación de técnicas de caja negra o técnicas de prueba funcional a aquellas donde los casos de prueba se derivan de la especificación.

Se resumirán a continuación algunas de estas técnicas[8]:

2.4.1.1 Partición de Equivalencia

Un caso de prueba es exitoso cuando detecta errores [Mye04]. Un caso de prueba es bueno cuando tiene una probabilidad razonable de encontrar un error. Dado que las pruebas se limitan a intentar con un subconjunto pequeño de todas las entradas posibles, esta técnica intenta seleccionar el subconjunto con la probabilidad más alta de encontrar errores.

La técnica consiste en subdividir el dominio de entrada en un número finito de clases de equivalencia, apoyándose para ello en la especificación. De cada clase se escoge un conjunto de representantes (puede ser solo uno) como entradas para la prueba, asumiendo que probar con dichos representantes es equivalente a probar cualquier otro valor de la clase. Es importante considerar tanto clases válidas como clases no válidas.

2.4.1.2 Análisis del valor límite

La experiencia dice que los casos de prueba que exploran condiciones límites producen mayor rentabilidad que aquellas que no lo hacen. Las condiciones límite ocurren en los bordes, por encima y por debajo de las clases de equivalencia.

[8] La clasificación no es taxativa. Se basa en una fusión entre las clasificaciones presentadas en [ISQ07] y [SWE04].

También se derivan casos de prueba considerando el espacio de resultados, es decir, considerando las clases de equivalencia de la salida [Mye04]. La técnica consiste en tomar valores en dichos límites como datos de prueba.

2.4.1.3 Tablas de decisión

Las tablas de decisión representan relaciones lógicas entre las condiciones (entradas) y las acciones (salidas). Los casos de prueba son derivados sistemáticamente considerando cada combinación posible de condiciones y de acciones [SWE04].

Presentan las siguientes ventajas [ISQ07]:

- Ayudan a capturar requerimientos que contienen condiciones lógicas.
- Pueden ser utilizadas para registrar reglas de negocio complejas.
- Ayudan a documentar el diseño interno.

El procedimiento consiste en analizar la especificación, para identificar las condiciones y acciones del sistema. Estas condiciones y acciones se representan por valores de falso o verdadero ("*booleano*s"). Se arma una tabla donde cada columna corresponde a una regla del negocio definitoria de una combinación única de las condiciones que la disparan, dando lugar a las acciones asociadas. La fortaleza consiste en crear combinaciones de condiciones que podrían no ejercitarse de otra manera [ISQ07].

2.4.1.4 Máquinas de estado finito

Un sistema puede exhibir diferentes respuestas dependiendo de las condiciones actuales o de la historia anterior (su estado) Este aspecto del sistema puede ser representado como máquinas de estado. Esta manera de modelar permite apreciar el *software* en términos de sus estados, las transiciones entre los estados, las entradas o las transiciones (los acontecimientos que disparan el cambio de estado) y las acciones que pueden resultar de esas transiciones. Los estados del sistema son identificables y finitos en número. Una tabla de estado muestra las relaciones entre los estados y las entradas y puede poner de relieve aquellas transiciones que no son válidas. Las pruebas se diseñan para cubrir una secuencia típica de estados, para cubrir cada estado, para ejercitar cada transición, para ejercitar secuencias específicas de transiciones o para probar transiciones inválidas. La técnica es sumamente usada en la industria del *software* embebido y autómatas, sin embargo es conveniente también para modelar un objeto del negocio que tiene estados específicos o flujos de diálogo de pantallas [ISQ07].

2.4.1.5 Grafo Causa Efecto

Una limitante de los métodos del valor límite y de la partición en clases de equivalencia es que no exploran combinaciones de los valores de la entrada. La

prueba de combinaciones de entrada no es simple debido a que el número de combinaciones puede ser astronómico. Se debe seleccionar un conjunto apropiado de subconjuntos de entrada. El grafo Causa-Efecto ayuda a seleccionar, de una manera sistemática, los casos de prueba. Es un lenguaje formal al que se traduce una especificación en lenguaje natural [Mye04]. Resulta en una técnica efectiva para analizar y representar relaciones lógicas [Bin99].

Una causa es una condición de entrada o una clase de equivalencia de las condiciones de la entrada. Un efecto es una condición de salida o una transformación del sistema. A partir de la especificación, se identifican las causas y los efectos. El contenido semántico de la especificación es transformado en un grafo "*booleano*" que vincula las causas con los efectos. El grafo así obtenido[9] se convierte en una tabla de decisión donde cada columna en la tabla representa un caso de prueba. Este aspecto presenta dificultades, aunque es un proceso algorítmico y automatizable [Mye04].

2.4.1.6 Casos de Uso

Los casos de uso capturan requerimientos funcionales, suministrando una narrativa acerca de cómo se usa el sistema mediante la especificación de una secuencia de acciones entre un actor y el sistema. Un actor es una entidad (alguien o algo) fuera del sistema que interactúa con él.

Tienen asociadas pre y poscondiciones. Las precondiciones son aquellas condiciones que deben cumplirse para que el caso de uso se realice exitosamente y las poscondiciones están constituidas por el estado final del sistema y los resultados, al finalizar el caso de uso. Tienen un escenario (flujo) principal y pueden tener varios escenarios alternativos. Las pruebas se pueden especificar a partir de casos de uso o escenarios. Los casos de uso describen los escenarios a través de un sistema basado en su uso probable, por lo cual los casos de prueba derivados a partir de ellos son muy útiles para encontrar defectos en el flujo durante el uso real del sistema. Resultan útiles para diseñar las pruebas de aceptación con la participación del cliente o del usuario y ayudan a descubrir defectos en la integración causados por la interacción e interferencia de diversos componentes [ISQ07].

2.4.1.7 Especificaciones formales

La especificación en un lenguaje formal permite la derivación de los casos de prueba en forma automatizada al tiempo que provee de una salida para referenciar y un oráculo para comprobar los resultados. Los casos de prueba pueden ser derivados de modelos (basados en modelos) o desde especificaciones algebraicas [SWE04].

[9] Su representación gráfica puede resultar a menudo intratable [Bin99].

2.4.2 Técnicas de Caja Blanca

Se denominan técnicas de caja blanca o técnicas de prueba estructural, aquellas técnicas donde los casos de prueba se derivan a partir de la estructura del sistema. Requieren conocer el código del programa a probar.

Se resumirán a continuación dos de estas técnicas:

2.4.2.1 Basadas en el flujo de control

El objetivo es cubrir todas las sentencias o bloques de sentencias en un programa, o combinaciones especificadas de ellas. Entre los múltiples criterios de cobertura el más fuerte es el de flujo de control, el cual ejecuta todas las trayectorias del flujo de control de la entrada a la salida. La prueba de trayectoria no es generalmente factible debido a los bucles, por lo cual se aplican en la práctica otros criterios no tan rigurosos, como ser: cobertura de sentencia, de decisión y de condiciones. La adecuación de tales pruebas se mide en porcentajes; se dice, por ejemplo, que se alcanzó una cobertura de sentencia del 100% si todas las sentencias han sido ejecutadas por lo menos una vez por las pruebas [SWE04].

2.4.2.2 Mutantes

Un mutante es una versión levemente modificada del programa a probar, diferenciado por un cambio sintáctico pequeño[10]. Cada caso de prueba ejercita el programa original y todos los mutantes. Si un caso de prueba permite identificar la diferencia entre el programa y un mutante, se dice que el mutante fue "matado" [SWE04].

Originalmente concebida como una técnica para evaluar un conjunto de casos de prueba, puede ser usada como criterio de prueba en sí misma. En el primer caso la proporción de mutantes muertos con respecto al total de mutantes generados puede ser una medida de la efectividad de las pruebas ejecutadas. En el último caso, las pruebas se diseñan específicamente para matar a mutantes que sobreviven. Para que la prueba sea eficaz, se deben derivar automáticamente y de una manera sistemática una gran cantidad de mutantes [SWE04].

2.4.3 Técnicas basadas en la experiencia o en la intuición

Existen técnicas que se basan en la intuición o la experiencia de quien lleva adelante las pruebas. Dentro de estas técnicas se encuentran las técnicas ad hoc, la prueba por conjetura de errores y la prueba exploratoria, las cuales se describen a continuación:

[10] Es aconsejable un solo cambio por mutante.

2.4.3.1. *Ad hoc*

La prueba "*Ad hoc*" es quizás la técnica más practicada. Las pruebas en este caso son derivadas confiando en la intuición, la habilidad y experiencia con programas similares [SWE04].

2.4.3.2. Conjetura de errores

Existen personas que parecen tener una particular destreza para encontrar errores, aún sin usar ninguna metodología en particular. Para hacerlo, conjeturan, usando la intuición y la experiencia, acerca de ciertos tipos probables de errores, escribiendo casos de prueba para exponerlos. Una forma de aplicar esta técnica es enumerar una lista de errores o identificar aquellas cosas omitidas por la especificación para las cuales seguramente el programador tuvo que asumir algo [Mye04].

2.4.3.3. Prueba Exploratoria

Kaner introdujo el término "prueba exploratoria" (*exploratory testing, "*ET") para referirse a una técnica donde se ejecutan las pruebas a medida que se piensa en ellas, sin invertir demasiado tiempo en prepararlas, confiando en los instintos. Realiza en simultáneo el aprendizaje, el diseño de los casos de prueba y la ejecución de las pruebas. Se utiliza la información obtenida mientras se prueba para diseñar más y mejores pruebas [Bac03].

La prueba exploratoria es aconsejable cuando:

- Se requiere obtener retroalimentación rápida de cierto producto o funcionalidad.
- Se necesita aprender el producto rápidamente.
- Se desea investigar y aislar un defecto.
- Se quiere investigar el estado de un riesgo.
- Se desea evaluar la necesidad de diseñar pruebas para esa área.
- No es obvio cual es la próxima prueba que debe realizarse.

Una estrategia para realizar pruebas exploratorias, según Kaner, es el principio del "*tour bus*": tener un plan general, pero permitirse desviarse de él por cortos períodos de tiempo, al igual que las personas que hacen un recorrido turístico en un bus y conocen los alrededores. La clave es no perderse el recorrido entero [Bac03].

La técnica conocida como "*Session Based Test Management*" de James Bach [Bac00], es una forma de prueba exploratoria que mantiene muy poca documentación. Es de bajo costo y medible pero requiere de verificadores experimentados y preparados.

Es de destacar que Bach explica en [Bac03] que la prueba exploratoria también es conocida como *Ad hoc*, pero dado que esta se asocia a connotaciones negativas comenzaron a utilizar el termino "exploratorio" en su lugar. El grupo de expertos que dieron origen a la "Escuela de Pruebas Guiadas por el Contexto" (*Context-Driven School of Software Testing*) [CDT05] (3.2.1) desarrolló a partir de allí esta práctica de prueba sin guión de forma que pudiera ser enseñable y tan disciplinada como cualquier otra actividad intelectual.

2.5 Patrones

Según la definición de Christopher Alexander[11] citada por Erich Gamma et al. en [Gam94]: "*Un patrón es una descripción de un problema que ocurre una y otra vez en nuestro entorno y la base de la solución a dicho problema, de forma que pueda usarse un millón de veces, sin tener que hacerlo de la misma forma dos veces*" (Christophen Alexander).

Los patrones constan de cuatro elementos esenciales: un nombre, el problema, la solución y las consecuencias de usarla. Los patrones permiten comunicarse y diseñar a un alto nivel de abstracción, estableciendo un vocabulario común de nombres, así como sistematizar la solución de problemas en ciertos contextos, brindando una plantilla que puede ser aplicada en diversas situaciones, en conocimiento de ciertas consecuencias derivadas de su implementación.

En la bibliografía de detallan numerosos patrones para las pruebas, organizados y clasificados de diferentes formas. Robert Binder [Bin99] propuso y detalló acerca de 37 patrones para el diseño de pruebas en OOLP, 17 patrones relativos a la automatización de las pruebas y 16 patrones referidos a oráculos. En la literatura e Internet se ha destacado acerca de la importancia de estos aportes de Binder dentro de su propuesta de pruebas basadas en modelos para ambientes de programación orientados a objetos. Un ejemplo de esto es el patrón *Server Stub* [Bin99] como antecedente de *Mocks* (4.3.2), según citan sus proponentes en [MFC01]. En cuanto a la prueba unitaria, Kent Beck, en [Bec02], propuso 6 patrones relacionados a la técnica de diseño *Test Driven Development* (TDD) (3.3), 20 relacionados a la prueba unitaria y 6 referidos al marco de automatización de pruebas xUnit (4.2). Marc Clifton, por su parte, publicó en [Cli04] una lista de 27 patrones, en tanto que Gerard Meszaros compiló y organizó 69 patrones, los cuales expuso y detalló en [Mes07], en especial para la utilización de las herramientas de automatización de la familia xUnit. Se entiende necesario destacar que, como mínimo, los diferentes conjuntos de patrones descritos se intersectan.

[11] Christopher Alexander, arquitecto, influenció notablemente el diseño en muchas disciplinas e inspiró el movimiento de patrones de software, con sus dos publicaciones: "*The Timeless Way of Building*", *Oxford University Press*, 1979 *y "A Pattern Language: Towns, Buildings, Construction* ", *Oxford University Press*, 1977.

A continuación se brindarán diagramas que ilustran acerca de algunos de los patrones básicos para pruebas unitarias según Clifton [Cli04]. Un análisis más pormenorizado de estos tópicos se considera que excede el alcance de esta tesis.

Patrón Simple de Pruebas (*Simple Test Pattern*)

Es el más simple. Se utiliza para comprobar si el código funcionará o fracasará para los valores de entrada suministrados.

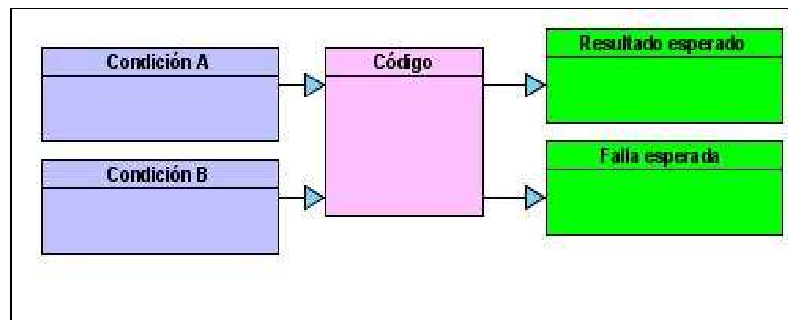


Fig.2- 1: Patrón simple [Cli04].

Patrón Simple de Pruebas con Datos (*Simple Test Data Pattern*)

Se separa el código que efectúa la prueba de los datos suministrados para la misma, con el objetivo de parametrizarla y poder reutilizar el código. Corresponde al tipo de pruebas impulsadas por datos (4.1.4). Los datos necesarios para preparar y ejecutar el código bajo pruebas se obtienen desde una fuente externa. Para verificar los datos también se recurre a comprobar contra los valores obtenidos desde la fuente externa.

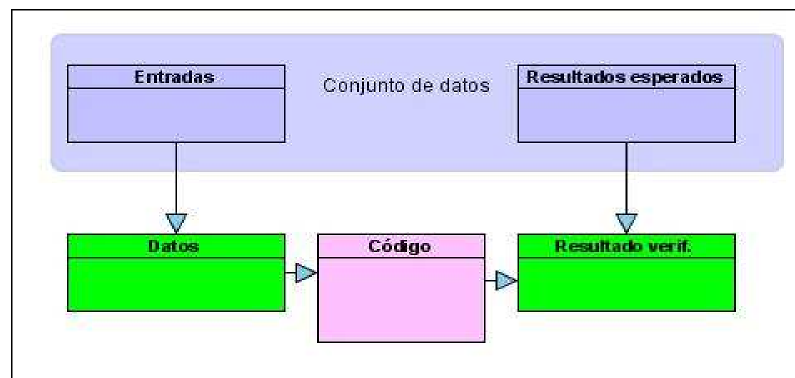


Fig.2- 2: Patrón simple con datos [Cli04].

Patrón Colección (*Collection Order*)

Se ingresa una colección y se comprueba el resultado contra una lista. Se ubica dentro de un conjunto más general de patrones que Clifton denomina "*Collection Management*".

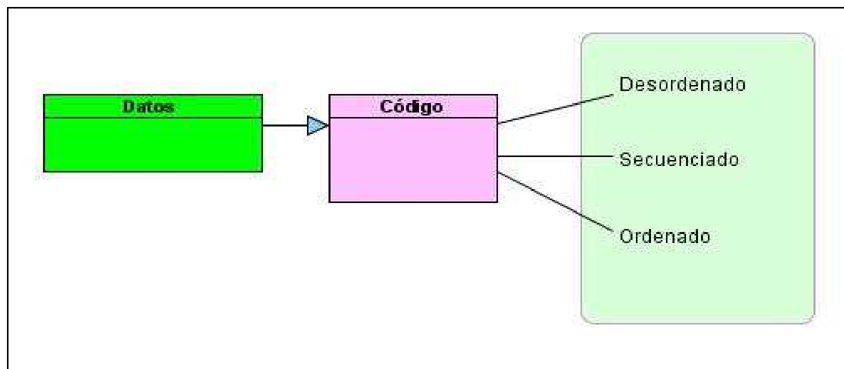


Fig.2- 3: Patrón colección [Cli04].

Patrón Proceso en secuencia *(Process-Sequence)*

Se utiliza cuando se desea comprobar procesos bajo determinada secuencia. Clifton lo ubica dentro de un conjunto más general de patrones que denomina "*Process*".

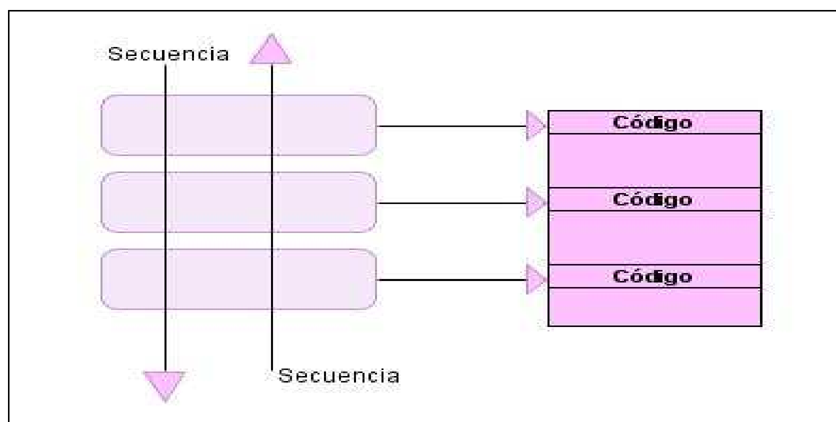


Fig.2- 4: Patrón proceso en secuencia [Cli04].

C a p í t u l o 3

LA VERIFICACIÓN Y VALIDACIÓN EN LAS METODOLOGÍAS ÁGILES

En este capítulo se presenta una visión de la V&V desde la óptica de las metodologías ágiles [MA01]. Se resumen las recomendaciones efectuadas por cuatro de las principales corrientes metodológicas de esta escuela y se estudia la disciplina de diseño "*Test Driven Development"* (TDD) [Bec02].

Está organizado de la siguiente manera:

- En la sección 3.1 se introduce a las metodologías ágiles.
- En la sección 3.2 se revisan los procesos, prácticas y roles recomendados con relación a la verificación y validación.
- En la sección 3.3 se estudia la disciplina de diseño "*Test Driven Development"* (TDD).

3.1 Metodologías ágiles

El conjunto de las propuestas metodológicas conocidas como "Metodologías Ágiles" surge a la luz a partir de la publicación del "*Agile Software Development Manifesto*"[12] [MA01] por parte de un grupo de expertos en febrero del año 2001, en *Salt Lake City, Utah*, intentando dar una respuesta al problema de lograr más rápidos y eficientes métodos para desarrollar *software*, focalizándose en los conceptos de simplicidad, velocidad y comunicación [Abr02] [Abr03] en contraposición a otras metodologías más predictivas orientadas a los procesos y la documentación. Se abordará en esta sección y la siguiente un resumen acerca de los aspectos comunes a este conjunto de metodologías y la visión de las pruebas desde su óptica en general y para cuatro de ellas en particular, abarcando el alcance de este análisis tanto a enfoques dinámicos como a enfoques estáticos. Un tratamiento pormenorizado de estas metodologías se considera fuera del alcance de este trabajo, existiendo en la literatura e Internet numerosa información sobre las mismas[13].

3.1.1 Características Generales

Los atributos básicos que caracterizan a estas metodologías pueden resumirse en los siguientes: alta cooperación, énfasis centrado en las personas y sus relaciones, adaptabilidad, procesos iterativos breves, desarrollo incremental, captura emergente de requerimientos (como opuesto a predictivo), empíricas (como opuesto a definidas), receptividad al cambio y documentación mínima.

Las ideas y prácticas que las sustentan no son novedosas [LB03] [Fow05] [Lar03]. Han sido aplicadas con éxito en empresas de otras industrias [TN86]. En particular la metodología EVO, propuesta por Tob Gib en 1960, presentaba características comunes a las metodologías ágiles [Lar03]. La novedad consiste en su aplicación conjunta y formalizada para el *software*.

Se apoyan en 12 principios [MAP01] que recogen los siguientes valores, enunciados en el Manifiesto Ágil:

- Individuos e interacciones predominando sobre procesos y herramientas.
- Programas operativos predominando sobre documentación extensa.
- Colaboración con el cliente predominando sobre la negociación de contratos.
- La respuesta al cambio predominando sobre el seguimiento de un plan.

Las corrientes que adhieren a estas metodologías se diferencian entre sí, básicamente, por proponer diferentes grados de ceremonia (el peso relativo de la documentación, pasos formales y revisiones) y por sus ciclos (la cantidad y duración de las iteraciones) [Lar03].

[12] Manifiesto del Desarrollo Ágil de *Software* (Manifiesto Ágil).
[13] En particular en el sitio "*Agile Alliance Library*" [AAL07] se ofrece un indice de gran cantidad de artículos en línea.

3.2 Verificación y Validación según las Metodologías Ágiles

Estas metodologías necesitan agilidad también al verificar y validar. Se presentaron importantes desafíos tales como los detallados a continuación:

- El desarrollo incremental, durante breves iteraciones, en base a requerimientos volátiles y documentación limitada, necesitó de la automatización de las pruebas. Esas mismas características convirtieron también en práctica fundamental a la prueba exploratoria (2.4.3.3).
- La determinación del alcance se constituyó en un aspecto crítico, enfatizándose en el análisis de riesgos.[14]
- La visión tradicional de muchos verificadores se vio perturbada al no contar con documentación detallada acerca del sistema a probar [Pet102].
- La necesidad de conciliar la existencia de equipos de verificación independientes con aquellos que aplican las metodologías ágiles para el desarrollo.
- La indefinición acerca de las responsabilidades y tareas de los verificadores en varias de las metodologías, con el énfasis puesto en las pruebas efectuadas por desarrolladores y usuarios.

La búsqueda de soluciones a estos y otros problemas dio lugar a varios aportes, como son los contenidos en [CH02], [BA04], [Jef00], [KBP01] y [Kni07]. La misión, tareas y responsabilidades de los verificadores fueron establecidas con mayor precisión y los procesos mejor abordados.

3.2.1 *Agile Testing*

Brian Marick menciona al "*Agile Testing*" [Mari01] como la práctica que sigue los lineamientos del Manifiesto Ágil aplicado a las actividades de pruebas. Esta práctica trata al desarrollo de *software* como cliente, define el rol del verificador en los proyectos ágiles, las actividades a realizar, los tipos de pruebas, la automatización y sus límites [Pet202] [Pet204].

Dicha práctica postula [Mari01] [Pet202] [Pet204]:

- <u>La comunicación durante el proyecto</u>**:** Abandonar la noción tradicional que la comunicación se establece básicamente con documentos de requerimientos y diseños hacia el verificador, quien responde con reportes de errores y planes de pruebas.

[14] El análisis de riesgos aplica en el área de las pruebas más allá del abanico de las metodologías ágiles. Múltiples autores han tratado el tema, entre ellos Bach [Bac99], Boehm & Turner [BT03], McGregor & Sykes [MS01]. Es una de las estrategias posibles para intentar alcanzar el objetivo de realizar la prueba menos costosa y que permita obtener la mayor efectividad posible.

- La integración de verificadores al equipo de desarrollo.
- La creación conversacional de casos de prueba en estrecha colaboración entre verificadores y clientes.
- El rol del verificador como entrenador en pruebas: Apoyar a clientes y desarrolladores en la especificación y ejecución de las pruebas y la búsqueda de defectos. Establecer parejas de desarrollador y verificador, de analista y verificador, de verificador y cliente. Participación en revisiones.
- El rol del verificador como nexo entre clientes y desarrolladores.
- La especificación mediante ejemplos con requerimientos ejecutables convertidos en pruebas, utilizando un formato inteligible a los clientes.
- Los planes de pruebas progresivos.
- Una alta automatización en las pruebas.
- La utilización de pruebas exploratorias.
- Las utilización de métricas e información provista por las pruebas, acerca del estado del *software* en desarrollo.
- La utilización del análisis de riesgos.

La corriente autodenominada "Escuela de Pruebas Guiadas por el Contexto" (*Context-Driven School of Software Testing*) [CDT05] fundada por Cem Kaner, Brian Marick, James Bach y Bret Pettichord en 1999 se relaciona estrechamente con "*Agile Testing*", al punto que algunos autores la consideran lo mismo, aunque al respecto James Bach sostiene que las metodologías ágiles serían uno de los contextos, dándole un rango mayor de acción [CDT07].

Sus planteamientos deben considerarse como un conjunto de valores. Se destaca que varios de sus proponentes han buscado en la epistemología y filosofía de las ciencias una visión sobre como pensar acerca de las pruebas [CDT07] [KBP01].

En el libro "*Lessons Learned in Software Testing*" [KBP01] sus autores presentan un considerable número de ejemplos desde la óptica de esta escuela, reunidos bajo la forma de lecciones aprendidas.

Los siete principios básicos de la Escuela son los siguientes [KBP01]:

- El valor de cualquier práctica depende de su contexto.
- Hay buenas prácticas en contexto, pero no existen las mejores prácticas.
- Las personas, trabajando juntas, son la parte más importante de cualquier contexto de proyecto.
- Los proyectos evolucionan en el tiempo de manera no siempre predecible.
- El producto es la solución, solo funciona si el problema es resuelto.
- Probar bien es un reto intelectual.
- Con juicio y habilidad, cooperando durante todo el proyecto, se podrá hacer lo apropiado en los momentos adecuados, para probar efectivamente los productos.

Tal como se aprecia varios de los valores comulgan con los enunciados de las metodologías ágiles, al hacer énfasis en el contexto, la incertidumbre y la comunicación; pudiendo ser compartidos fácilmente aplicando el sentido común. Por otra parte la mayoría de los enunciados escapan del área de pruebas para posicionarse en un nivel más general.

La puesta en acción de estos principios, aplicados a las pruebas, podría enunciarse de la siguiente manera [KBP01]:

- Los grupos de verificación no desarrollan el proyecto, le sirven.
- Las pruebas son hechas en representación de los involucrados con diferentes estrategias según los objetivos.
- Las métricas no válidas son peligrosas.
- El valor esencial de un caso de prueba es proveer de información.
- Todos los oráculos son falibles.
- Automatizar las pruebas no implica automatizar las pruebas manuales.
- Diferentes tipos de defectos serán revelados por diferentes tipos de pruebas.
- Las pruebas deben enfocarse en diferentes riesgos o ser más incisivas a medida que avanza el desarrollo.
- Los artefactos de prueba deben satisfacer los requerimientos más relevantes.

Entre las múltiples técnicas a utilizar aconsejan pruebas de "caja gris" (pruebas funcionales en conocimiento del código y aprovechando ese conocimiento) y "exploratorias" (2.4.3.3).

3.2.2 Aspectos relevantes en cada metodología

Se analizarán los criterios más relevantes, considerando tanto técnicas dinámicas como estáticas, para las siguientes metodologías:

- *eXtreme Programming*: (XP) (ver Anexo B)

  Sinopsis

  Propuesta por Kent Beck y otros [C3T98] se apoya en una colección de prácticas conocidas de la Ingeniería de *Software* enlazadas y alineadas para funcionar en conjunto, diseñada con el ánimo de crear programas exitosos a partir de requerimientos volátiles y vagos [Abr03]. Propone un proceso iterativo e incremental, prescribiendo iteraciones muy breves, fuerte involucramiento del cliente (quién proporciona sus requerimientos bajo la forma de narrativas breves conocidas como relatos o "*stories*"), integración continua y reconstrucción de código, fuerte actividad de pruebas, especialmente unitarias y de aceptación, automatización de las mismas, escritura de código para pruebas previo al código de producción (*Test First Programming*), pequeñas liberaciones. Se basa en 5 valores, 14 principios y

24 prácticas. Se presenta por parte de Kent Beck en [Bec199], [Bec299] reformulándose en [BA04]. Se formaliza sobre finales de la década de 1990. En el anexo (B) a este trabajo se la describe con más detenimiento.

Acerca de V&V

Hace fuerte hincapié en las pruebas, especialmente en las pruebas unitarias, de integración, regresión y aceptación. Beck y Andres sostienen al respecto que: "*en XP las pruebas son tan importantes como la programación*" [BA04] advirtiendo acerca que los defectos destruyen la confianza requerida para el desarrollo. Destacan que siendo imposible eliminar todos los defectos, también es excesivamente costoso intentar como paliativo incrementar el tiempo medio entre fallas en forma considerable, como ser pasar de un mes a un año. El dilema está en que las fallas son a su vez muy costosas, reconociéndose que el propósito para el cual se desarrolla cada sistema implica necesariamente diferentes niveles de aceptación. Dada esta realidad es que muchos de los principios y prácticas de XP se vinculan con las actividades de prueba y la disminución de la cantidad de defectos.

En cuanto a las prácticas, el escribir el código para implementar las pruebas unitarias previamente al código que debe salvarlas (*Test First Programming*), aunada a la práctica de reconstrucción de dicho código (*Refactoring*) ha dado lugar a la formalización de una disciplina de diseño que escapa a las fronteras del propio XP conocida como "*Test Driven Development*" (TDD) [Bec02] [Ast03] la cual se abordará con más detalle en la sección (3.3). La práctica de reconstrucción para mejorar el código sin alterar la funcionalidad, por otra parte, también requiere de la aplicación sistemática de pruebas de regresión, para verificar que el comportamiento de la unidad no cambió [Fow99]. Por último, la práctica de "programación en parejas" (*pair programming*) apoya la aplicación de TDD así como promueve otras técnicas de reconocida efectividad en la reducción de defectos y mejor diseño como ser la revisión entre pares (2.1.5.3) [CW01]. En este punto resulta interesante mencionar que Cem Kaner, James Bach y Bret Pettichord [KBP01] van un paso más allá y recomiendan extender el concepto de "programación en parejas" hacia las tareas efectuadas por los verificadores, para los cuales proponen una práctica similar, la "prueba en parejas", no circunscripta a las fronteras de XP. Para las pruebas de aceptación se enfatiza también en una estrategia de elaboración de las mismas desde etapas tempranas, previo a la existencia del código que ha de aprobarlas, dando lugar a una extensión de TDD aplicado a nivel funcional [Bec02] conocida como "*Acceptance* TDD" (ATDD) la cual se abordará en (3.3.1). Cada relato en XP debe ser verificable y se acompaña por sus pruebas de aceptación. Las pruebas de aceptación superadas sirven como métrica para cuantificar el avance del proyecto.

En cuanto a los principios, bajo "beneficio mutuo" se enfatiza la elaboración de pruebas automatizadas como vehículo de comunicación y documentación;

bajo "auto-semejanza" aplica TDD y ATDD; bajo "redundancia" la programación en parejas, TDD y ATDD; en "pasos pequeños" TDD e integración continua pasando la batería de pruebas [Tos07].

Beck y Andres [BA04] sostienen que la mejora de la efectividad y la disminución del costo en las actividades de pruebas se apoyan en dos pilares:

- Comprobación doble: Refiere a programar las pruebas y los programas que deben salvarlas para lograr la armonía entre unos y otros. Refiere también a los dos juegos de pruebas, uno de parte de los programadores y otro de parte de los clientes. Incluye también referencias a la importancia de pruebas estáticas automatizadas.
- Retorno rápido para evitar el incremento del costo de enmendar defectos (DCI) (*Defect code increase*): Cuanto más tarde se descubre un error más cuesta su reparación. El rápido retorno implica que las pruebas deben automatizarse.

Por último, la figura del verificador está prevista, estableciéndose que auxilia a los clientes en la elaboración de las pruebas de aceptación, encargándose de ejecutarlas regularmente. Mantiene las herramientas de prueba, optimiza los casos de prueba, divulga los resultados y apoya a los desarrolladores en sus pruebas con sesiones conjuntas cuando se requiera [BA04]. Al respecto, Crispin y House [CH02] definen a los verificadores como individuos que además de producir y ejecutar pruebas tienen habilidades que combinan las tareas de aseguramiento y control de la calidad. Los verificadores ven el sistema desde el punto de vista del cliente pero con la posibilidad de captar los detalles del sistema, sus posibilidades y sus restricciones. Tienen formación que les da la posibilidad de capturar mejor los defectos y ayudan al equipo a mantenerse en la dirección correcta. Actúan dentro de un equipo XP preservando los derechos de los clientes y siendo guardianes de los derechos de los desarrolladores. Inciden en la planificación brindando estimación de los tiempos para las pruebas y también la estimación de los tiempos para la corrección de defectos, entregando al cliente elementos que le permitan tomar decisiones más acertadas sobre el producto a desarrollar. De todas formas se destaca que es una característica en XP que los roles en los equipos "maduros" se complementen e intercambien.

- *Dinamic System Development Method* (DSDM) [DSDM107]

Sinopsis

Fue diseñada por un consorcio en el Reino Unido y se la señala como la primera metodología ágil en formalizarse [Abr03]. Está basada en las ideas del *"Rapid Application Development"* (RAD) de James Martin [LB03] a las que aplica un proceso estándar iterativo e incremental, donde las iteraciones tienen asignado en tiempo fijo. La idea más distintiva consiste en que se

deben fijar el tiempo y los recursos y luego, en base a ellos, la funcionalidad a obtener. Prescribe siete fases, con actividades y entregables para cada fase. Se basa en 12 principios, entre ellos colaboración, cooperación e involucramiento activo del cliente.

Acerca de V&V [15]

La metodología plantea los siguientes principios para las pruebas: [DSDM207]

- Validar a todo nivel que el sistema cumple su propósito, integrando las pruebas en todo el ciclo de vida.
- Priorizar la prueba de las funcionalidades fundamentales.
- El objetivo puesto en encontrar errores.
- Se prueban todos los productos a todos los niveles.
- Contar con un equipo independiente de verificación.
- Deben ser repetibles.

Cuando se utiliza DSDM en conjunción con las prácticas de la metodología eXtreme Programming (XP) (Anexo B) se agregan los principios de "escribir primero las pruebas" y "reconstrucción", equivalentes a TDD. En este caso aplican también las consideraciones efectuadas al tratar las pruebas bajo la óptica de XP.

DSDM enfatiza que las actividades de prueba deben priorizarse basadas en los objetivos del negocio y realizarse en todas las fases indicadas por la metodología durante todo el ciclo de vida. En las dos primeras (estudio de factibilidad y estudio del negocio) se deben evaluar las diferentes áreas riesgosas a probar y elaborar la estrategia de pruebas. Dicha estrategia debería ser el punto de referencia de todos los aspectos relacionados con las pruebas y debe revisarse previo a la entrada a cada iteración *(timebox)* junto al conjunto discreto de requerimientos para prueba asignados. Se ofrecen guías y recomendaciones para proyectos típicos acerca de estas actividades, dónde deberían cumplirse, los artefactos a producir, los roles intervinientes y los criterios de calidad general a satisfacer, pero debido a que cada proyecto es diferente debe elaborarse la estratégia para cada uno en particular. La metodología describe también un patrón de aplicación de los tipos de prueba según cada fase, estando mencionadas entre otras: pruebas unitarias, de enlace, funcionales del sistema, de "usabilidad", de aceptación, de instalación, de portabilidad, de aceptación operacional y un grupo específico de pruebas de "credibilidad" que aseguren ciertas certezas, basadas en la intuición, que apuntan a probar cada funcionalidad de la forma más fundamental. En las fases de elaboración del modelo funcional, diseño y construcción las pruebas

[15] Los manuales DSDM son privativos para los miembros del consorcio, por lo cual la presente descripción se basa en información de acceso libre brindada por el consorcio o en otros documentos accesibles en Internet.

deben ocurrir dentro de cada iteración, con técnicas dinámicas y estáticas según corresponda [DSDM04].

Dadas las restricciones de tiempo y recursos impuestas el análisis de riesgos se define como una actividad fundamental; recomendando la utilización de alguna forma del método denominado "*Risk Based Testing*" (RBT) (2.2.1.2). Las pruebas deben cubrir los aspectos que resulten críticos, para lo cual se identifican y evalúan los riesgos. Este análisis de riesgos da soporte a los entregables de estrategia, planes y especificación de pruebas. Los casos de prueba se priorizarán en base al riesgo y al impacto, con el objetivo que si el tiempo asignado expirara antes de culminar las pruebas, queden fuera solo las menos prioritarias. Declara, en virtud de las mismas restricciones, que se debe tener especial cuidado de no excederse en la documentación ni en el uso de técnicas formales de prueba, las que pueden no ser recomendables según sea la escala de tiempos a manejar[16]; se reconoce, sin embargo, que el plan de pruebas es indispensable aunque no con el detalle tradicional [DSDM207].

Dentro del conjunto de actividades de pruebas a realizar, se asigna a las pruebas estáticas (inspecciones y revisiones) un papel fundamental en la detección temprana de defectos y en la captura de conocimiento, por lo cual insta a realizarlas, tanto de modo formal como informal, buscando el balance apoyándose en el RBT y bajo las restricciones de tiempo y costo, mencionándose la utilidad de los analizadores estáticos. Se establece que deberían automatizarse todos los casos de pruebas hechos por los desarrolladores, de regresión y de credibilidad, si las escalas de tiempo lo permiten. De usarse en conjunción con XP se establece el uso de TDD.

En cuanto a roles, el "embajador" es responsable sobre las pruebas de usuarios. El verificador se integra al equipo de desarrollo. Un miembro clave de cada equipo podrá cumplir, de ser necesario, el rol de coordinador de pruebas del proyecto. La estrategia general y las actividades de pruebas a nivel de varios proyectos deberán ser aseguradas y coordinadas por el administrador de pruebas (*Test Manager*). Se aconseja que los usuarios se involucren, con sesiones conjuntas con el personal técnico [DSDM04].

- Feature Driven Development (FDD) [FDD06]

  Sinopsis

  Orientada por procesos y pensada para construir sistemas críticos de negocio, propone un proceso iterativo e incremental, con 5 procesos secuenciales donde los dos últimos se ejecutan en forma iterativa, diseñando y

[16] El tiempo para DSDM es una restricción, se fija y en base al mismo se evalúa lo que es posible hacer; por lo cual advierte que técnicas formales como ser la partición en clases de equivalencia probablemente no deban aplicarse.

construyendo por cada funcionalidad (*feature*)[17] en ciclos semanales. Se pone el foco en el diseño y construcción, con intenso monitoreo y actividades de inspección [Abr03]. Se definen los procesos y los criterios de salida de cada uno. Fue desarrollada por Peter Coad, Jeff De Luca y Stephen Palmer a partir de 1998.

Acerca de V&V

Hace énfasis en las revisiones e inspecciones, tanto para el código como para el análisis y diseño, prácticas de muy alta efectividad en la detección temprana de defectos. Recomienda aplicarlas en los procesos prescriptos por la metodología [FDD07], especialmente en los denominados:

- Proceso 1: desarrollar un modelo general (*develop an overall model*).
- Proceso 4: diseñar por funcionalidad (*design by feature*).
- Proceso 5: construir por funcionalidad (*build by feature*).

También pone el acento en la prueba unitaria, práctica obligatoria en el proceso quinto, aunque no se estipula que deba regirse por TDD u otra técnica en particular. Se sostiene, sin embargo, que puede acoplarse con TDD sin mayores dificultades, ajustando los procesos cuarto y quinto y sus criterios de salida [FDD03]. En tanto, la "programación defensiva", como ser la validación de parámetros de entrada, es aconsejada para evitar defectos de frecuente aparición.

Jeff De Luca [FDD05] explica que por lo menos hay tres niveles de pruebas que pueden encajar dentro del proceso quinto, "*build by feature*" (BBF): "*class-based testing*" (pruebas unitarias), "*feature against other features"* (pruebas de integración) y "*dev test*" (pruebas por funcionalidad). Las clases pueden ser promovidas para la construcción del sistema (*build*) solo luego de pruebas unitarias e inspecciones de código satisfactorias.

En cuanto al rol del verificador, se lo estipula como un rol adicional. Su cometido es verificar que el sistema se ajuste a los requerimientos de los clientes. Puede estar integrado al equipo o constituir parte de un equipo independiente.

También se estipula un rol de soporte (*toolsmith*) que tiene entre sus tareas construir pequeñas herramientas para pruebas. Se destaca, sin embargo, que un rol puede ser compartido por varias personas y que una persona puede asumir varios roles [Abr02].

[17] Una "*feature*" es una pequeña pieza de funcionalidad con valor para el cliente. Para cada una se ingresa en breves ciclos de planificación, diseño y construcción.

- Scrum [Scr07] [SB01]

  Sinopsis

  Esta metodología fue desarrollada para administrar actividades en un ambiente volátil. Es un acercamiento empírico basado en flexibilidad, adaptabilidad y productividad. Reconoce al proceso de desarrollo como una actividad caótica, establece controles y se apoya en equipos autogestionados. Es una metodología enfocada en la gestión, la cual se puede definir en términos de patrones organizacionales [Bee97] [Bee98]. Especifica básicamente 3 roles[18], 3 tipos de reuniones (planificación, reunión diaria, reunión de revisión) y 3 artefactos (lista de requerimientos priorizada general o "*backlog",* lista de requerimientos priorizada para cada iteración y gráfica de control del tiempo remanente o "*burndown chart*"[19] [Ols07]. Los ítems del *backlog* se asignan a iteraciones de 30 días. Permite que se escojan las técnicas, métodos y prácticas que se juzguen más convenientes para el proceso de implementación. Fue formulada para procesos de desarrollo de *software* por Ken Schwaber, Michel Beedle y Jeff Shuterland a partir de 1993 [Sch95] [ADM96] en base a ideas de Ikujiro Nonaka e Hirotaka Takeuchi [TN86] alimentándose también de ideas tomadas de Weinberg y de la "Teoría de las Restricciones" de Goldrat.

  Acerca de V&V

  Estando primordialmente enfocada en la gestión de proyectos, ofrece poca información acerca de las pruebas de programas y de los roles involucrados; sin embargo es perfectamente posible, dentro de la misma, integrar otras prácticas, en particular de eXtreme Programming [Kni07] por lo que aplican en ese caso, además de las consideraciones que se realizarán, aquellas efectuadas al analizar la prueba de programas en XP. Se destaca que dada la característica de Scrum, este podría ser utilizado, ajustándole, como marco para la gestión de procesos de pruebas [Ols07].

  Según se indica en la descripción de la metodología [Sch95] [Scr07] [SB01] las pruebas unitarias y de integración se realizan dentro de cada iteración (*sprint)*, recomendándose el uso de herramientas de automatización y la aplicación de TDD. Las pruebas funcionales y otras se realizan sobre el final de cada iteración y eventualmente durante las siguientes, pudiéndose también aplicarse las mismas técnicas de gestión y control en iteraciones para pruebas (o ciclos de pruebas) paralelos con el desarrollo. Se utilizan iteraciones de estabilización del producto, abocadas a actividades de prueba y corrección de defectos. Se prevén instancias de demostración del producto a clientes y otros

[18] De considerarse al equipo de desarrollo visto en su totalidad como "un rol".

[19] Los proponentes de esta metodología consideran inapropiado utilizar técnicas tradicionales como ser diagramas de Gantt o Pert para la gestión de los proyectos de *software*. Proponen en cambio un mecanismo adaptativo de repriorización permanente sobre los ítems de la lista priorizada y como mecanismo de control gráficas sobre el tiempo de trabajo remanente y los ítems cumplidos.

involucrados al final de cada iteración para requerir su aprobación en general (aceptación) y mostrar las funcionalidades logradas. Los ítems concernientes a la prueba se adicionan a la lista priorizada (*backlog*) para su correcta identificación, priorización y estimación.

Se han realizado varios aportes metodológicos basados en lecciones aprendidas. Entre otros en [Gal07], [Kni07] y [Koh05] se proponen soluciones a varios problemas que se han presentado en la práctica de Scrum, especialmente para las pruebas que escapan al alcance de una iteración, como ser aquellas que deben ser realizadas por los clientes para aceptar los entregables, así como para las pruebas dentro de cada iteración y la coordinación con equipos de pruebas independientes. Se plantean opciones tales como efectuar iteraciones de duración diferente para las actividades de estabilización [Gal07] y un equipo de verificación que intercepte y efectúe pruebas antes de brindar el producto al cliente para sus pruebas de aceptación Es muy recomendable estructurar las iteraciones teniendo en cuenta las actividades de prueba a desarrollar, que deben ser consideradas en la lista priorizada, siendo cuidadosos pues a medida que avance el proyecto es altamente probable que se recargue la actividad de los verificadores y del equipo en general, una vez que se comienza a obtener el retorno dado por los resultados de las pruebas [Kni07].

Es altamente recomendable integrar verificadores en el equipo de desarrollo estableciendo canales para una muy frecuente comunicación, que participen en las reuniones (s*crum meetings*) y utilicen técnicas exploratorias como fuente de información; así como la lista priorizada (*backlog*) tanto para la información como para la planificación y elaboración de las pruebas [Kni07] [Koh05].

Se recomiendan como tareas posibles para los verificadores: apoyar a los programadores en sus pruebas unitarias, montar la estructura de pruebas, clarificar requerimientos y asistir a los clientes en las pruebas de aceptación [Kni07].

3.3 *Test Driven Development*

Test Driven Development (Desarrollo Dirigido por Pruebas) es una disciplina de diseño y programación, donde cada nueva línea de código es escrita en respuesta a una prueba que ha fallado. Sin ser definida como una metodología de pruebas se apoya fuertemente en las pruebas unitarias y también en las pruebas de aceptación, basándose en prácticas formalizadas por *eXtreme Programming*. Ha despertado gran interés tanto en ámbitos académicos como en la industria, encontrándose en la literatura investigaciones sobre su aplicación, aunque aún en limitado número, ya sea mediante experimentos controlados o reportes acerca de su práctica en la industria.

Esta sección abordará *Test Driven Development* en general, el dominio de su aplicación, consideraciones y algunas evidencias empíricas y experimentos que evalúan los resultados de su aplicación. Recoge el reporte técnico "*Test Driven Development*. Fortalezas y Debilidades" [Ara07].

3.3.1 Características

Test Driven Development es una disciplina iterativa e incremental de diseño y programación, donde cada nueva línea de código se escribe en respuesta a una prueba fallida que el propio programador ha programado. Es parte medular del desarrollo ágil de código derivado de XP, donde resulta en una práctica crítica, y de los principios del Manifiesto Ágil.

Apoyada en la popularidad de XP, *Test Driven Development* ha tenido gran difusión mundial en los últimos años, siendo posible adoptarla dentro de cualquier otro tipo de proceso de desarrollo de *software*. Información sobre experiencias de utilización de TDD han sido recientemente publicadas, entre otros, en [FIT307], [JM07], [Mel07], [MM07], [Rea05] y [Sin06].

Su principal objetivo puede resumirse en obtener "*código limpio que funcione*"[20] [Bec02].

Siguiendo la categorización citada en [Mug03], TDD puede ser aplicado a dos niveles, a saber:

- Nivel de micro-iteraciones:

  En este nivel el desarrollo es guiado por pruebas unitarias escritas por el propio programador de la aplicación, donde los pasos son, según [Bec02]:

  - Rápidamente agregar una prueba.
  - Correr todas las pruebas y comprobar que solo la nueva falla.
  - Hacer un pequeño cambio.
  - Correr todas las pruebas y comprobar que pasan en forma satisfactoria.
  - Reconstruir para remover duplicaciones.

  Lo expuesto se puede resumir en el diagrama de actividad de la figura 3-2.

[20] Frase acuñada por Ron Jeffries.

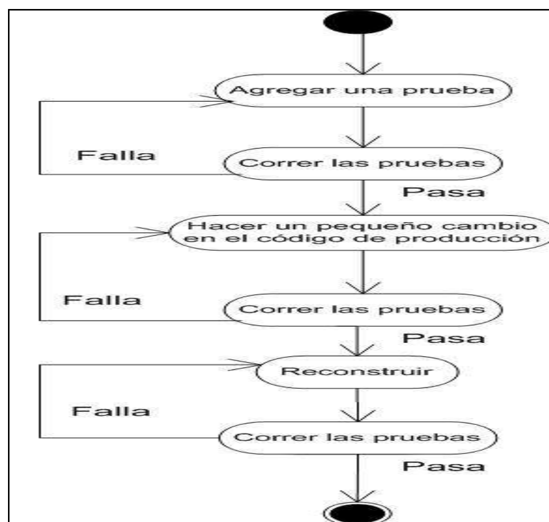


Fig. 3- 1: Pasos en un ciclo TDD.

Las principales tareas se pueden agrupar en "Escribir y ejecutar pruebas", "Escribir código de producción" y "Reconstrucción", detallándose a continuación:

Escribir y ejecutar pruebas:

La escritura de las pruebas sigue varios patrones identificados y detallados por Beck en [Bec02]. En particular se destaca que las pruebas:

- Son escritas por el propio desarrollador en el mismo lenguaje de programación que la aplicación, y su ejecución se automatiza. Esto último es primordial para que obtenga un retorno inmediato, indicándose que el ritmo de cambio entre prueba-código-prueba está pautado por intervalos de no más de diez minutos. La automatización también permite aplicar, en todo momento, la batería de pruebas, implicando pruebas de regresión inmediatas y continuas.
- Las pruebas se deben escribir pensando en primer lugar en probar las operaciones que se deben implementar.
- Deben ser aisladas, de forma que puedan ejecutarse independientemente de otras, no importando el orden. Este aislamiento determinará soluciones de código cohesivas y de bajo acoplamiento, con componentes ortogonales[21].
- Deben escribirse antes que el código que se desea probar.

Escritura de código:

La escritura de código es el proceso por el cual se logra hacer que la prueba unitaria pase [Sin06]. La propuesta de Beck contempla cuatro estrategias para

[21] En el sentido de independencia entre objetos. En el libro "*The Pragmmatic Programmer: From Journeyman to Master*" se expone una definición y una discusión acerca de la importancia de dicho tópico [HT99].

salir rápidamente de la situación de falla, descritas como patrones en [Bec02]: Es Falso (*Fake It*): Retornar un valor esperado, Triangulación, Implementación obvia y Uno a muchos. En cuanto a la programación en si, la recomendación consiste en programar por intención (*programming by intention*), lo cual implica pensar y expresar en el código la intención, más que los algoritmos [Jef00].

Reconstrucción (*Refactoring*):

En Fowler et al. [Fow99] se define la reconstrucción como una practica que consiste en mejorar el código existente, de manera disciplinada, sin alterar su comportamiento externo, para hacerlo más fácil de entender y modificar. La mejora de código se logra mediante la aplicación de una serie de consejos entre los cuales de destaca la eliminación de duplicación, la división en segmentos menores, la estructuración de manera que resulte más conveniente y la aplicación de las ventajas que brinde el lenguaje de programación[22]. La reconstrucción cumple un rol sustancial como complemento del diseño. El orden de ejecución de las tareas se resume con la frase Rojo/Verde/Reconstrucción (Red/Green/Refactor). En Rojo se está en la etapa en que una prueba falló, en Verde en la etapa cuando la aplicación pasa las pruebas.

A modo de resumen, se establece que TDD a este nivel sigue tres grandes leyes [Mar07]:

- No se escribe código de producción salvo que se hubiera escrito previamente una prueba unitaria que falló.
- No se escribe más de una prueba unitaria que sea suficiente para fallar.
- No se escribe más código de producción que el suficiente para pasar la prueba unitaria

- Nivel Iteración o Funcional:

El desarrollo es guiado por pruebas de aceptación. Este nivel, mencionado en [Bec02] como ATDD (*Acceptance TDD*), consta de los siguientes pasos:

- El usuario especifica pruebas antes que las funcionalidades sean implementadas.
- Una vez que el código es escrito la prueba sirve como un criterio de aceptación.

La construcción de las pruebas es también a este nivel un proceso evolutivo, salvo que el retorno se da en ventanas de tiempo más extendidas, a nivel de iteración, y están fundamentalmente en manos del usuario apoyado por verificadores y

[22] En particular referido a facilidades de la programación Orientada a Objetos (p/ej: Polimorfismo y Herencia).

programadores. Dado que se requiere la automatización de las mismas se lo conoce también como "*Executable Aceptance TDD*" (EATDD) o "*StoryTest-Driven Development*".

En *Test Driven Development* las pruebas siempre actúan, en ambos niveles, como estímulos para el desarrollo; el cual responde para satisfacerlas. Tal como se expresa en [Bec02] el punto de vista de las pruebas por parte de TDD es pragmático, son estas el medio utilizado para obtener certezas, vencer los temores y guiar el desarrollo.

3.3.2 Consideraciones

Se expondrán a continuación diversas consideraciones en relación a TDD.

3.3.2.1 Utilización de herramientas

Para viabilizar la utilización de TDD, especialmente a nivel de micro-iteración, se debe contar con herramientas que permitan la automatización de las pruebas. Beck advierte al respecto que "*una porción de código sin un programa que le pruebe automáticamente simplemente no existe*" [Bec199]. A nivel funcional es más dificultosa la automatización, especialmente pues la prueba puede ser asimilada a un ciclo funcional, lo que acrecienta la complejidad de los casos, con altamente probable intervención de interfases, especialmente gráficas (GUI), desde las cuales la aplicación interactúa con el usuario.

Las herramientas disponibles se especializan en general para pruebas unitarias o pruebas de aceptación, aunque la división no es estricta y podrían utilizarse en ambos niveles, sea mediante la incorporación de extensiones y bajo ciertas restricciones. En particular se han desarrollado varios marcos *open source* para pruebas, con la participación en su gestación de los proponentes principales del emergente TDD. Ejemplos de esto son la familia de marcos para pruebas unitarias xUnit (4.2) a nivel de la micro-iteración y el marco para pruebas de aceptación FIT a nivel funcional (*Framework for Integrated Tests*) (5).

3.3.2.2 Prevención de defectos

Según cita Melnik G. [Mel07], los encuestados en el reporte "*Evaluating ROI from Software Quality*" de *Cutter Consortium* identificaron a TDD como la práctica con el segundo nivel de impacto en la prevención de defectos, siguiendo a las inspecciones de código.

El diseño continuo de pruebas que aplica TDD está en línea con la aseveración de Beizer acerca que "*el acto de diseñar pruebas es de las técnicas más efectivas para la prevención de defectos*" [Bei90].

3.3.2.3 Aspectos sicológicos

Siendo las pruebas un proceso destructivo se establece una dificultad sicológica para la correcta elaboración de las pruebas por parte de los programadores [Mye04] (2.2.2). Dado que el programador que utiliza TDD debe codificar las pruebas primero, este método podría diluir la sensación que la prueba destruye el código. Se invierte la posición; bajo TDD es el código el que salva la prueba, convirtiéndose el acto de la prueba en una carrera de salvar obstáculos, paso a paso.

Por otra parte el programador obtiene un rápido retorno acerca del estado de salud de la aplicación. Si introduce mejoras en el código o nuevas funcionalidades, sabrá rápidamente si pasan todas las pruebas. Esto reduce la ansiedad y el temor [Bec02].

Por último, se debe tener en cuenta que los ciclos extremadamente cortos y el cambio entre pruebas y codificación, en el ámbito de la micro-iteración, podrían hacer que los programadores sientan a TDD como algo contrario a su intuición y de naturaleza agobiante.

3.3.2.4 Método

Según Mugridge el método científico puede ser explorado como una metáfora para *Test Driven Development*, detectando una correspondencia entre ambos, especialmente al considerar la evolución de la teoría, la selección de experimentos y el rol de la reproducibilidad en la experimentación [Mug03].

Más allá de esto en [Agu05] se discute acerca de la real utilidad práctica de TDD, específicamente durante la micro-iteración, cuando se efectúa el desarrollo con generadores de programas a partir de especificaciones realizadas en el nivel adecuado de abstracción.

3.3.3 Evolución

Entre los aportes que se han efectuado a la disciplina "*Test Driven Development*" se resumirá acerca de los siguientes:

3.3.3.1 "*Behaviour Driven Development*"[23] (BDD)

Es un refinamiento de TDD con énfasis en definir especificaciones de comportamiento ejecutables más que "pruebas", combinando las mejores prácticas recogidas por TDD con otras prácticas, a los efectos de solucionar impedimentos técnicos y organizacionales en la adopción de TDD [Ada07]. Fue propuesto originalmente por Dan North [Nor06] como solución a problemas que él detectó durante su experiencia en la práctica y enseñanza de TDD.

[23] En el contexto de este trabajo se lo traduce como "Desarrollo guiado por comportamientos".

Se basa en referirse a especificaciones y comportamientos en lugar de pruebas y unidades, encontrando su punto de partida en la propuesta de Eric Evan [Eva04] de creación del lenguaje ubicuo del dominio para el proceso de análisis. David Astels [Ast05] mencionó, entre los fundamentos teóricos que apoyan la propuesta, a la hipótesis de Sapir-Whorf[24] y a las corrientes del conocimiento acerca del papel del lenguaje en el desarrollo del pensamiento, la relación entre las categorías gramaticales del lenguaje que se utiliza y la comprensión del entorno. Dio lugar a proyectos tales como Jbehave [Jbh07].

3.3.3.2 “*Need Driven Development*”[25] (NDD)

Es una propuesta de Steve Freeman, Tim Mackinnon, Nat Pryce y Joe Walnes [Fre04], con el propósito de guiar el desarrollo descubriendo las interacciones entre objetos basándose en los roles. Se apoya en aplicar intensivamente una técnica denominada *Mock Objetcs* (4.3.3), extensión de TDD.

[24] http://en.wikipedia.org/wiki/Sapir-Whorf_hypothesis
[25] En el contexto de este trabajo se lo traduce como “Desarrollo guiado por necesidades”.

Capítulo 4

AUTOMATIZACIÓN DE LAS PRUEBAS

En este capítulo se examinan conceptos relativos a la automatización de las pruebas y se estudia acerca de la automatización de las pruebas unitarias. Se describe sobre la tecnología de código libre (*open source*) xUnit para la prueba automatizada en dicho ámbito. Se analizan técnicas para reemplazo, durante las pruebas, de unidades de las cuales el SUT depende y la relación de estas técnicas con la disciplina de diseño TDD.

Está organizado de la siguiente manera:

- En la sección 4.1 se trata en general acerca de la automatización de las pruebas y tecnologías relacionadas.
- En la sección 4.2 se estudia sobre la familia de herramientas de prueba unitaria xUnit.
- En la sección 4.3 se describe acerca de técnicas de reemplazo temporario de unidades de las cuales depende la unidad a probar y su relación con TDD.

4.1 Automatización

En esta sección se examinan conceptos relativos a la automatización de las pruebas, se brinda una clasificación de las herramientas, se enuncian objetivos y filosofía aplicables en el ámbito de la automatización de pruebas unitarias y de validación discutiéndose acerca de los beneficios, riesgos y limitaciones de su aplicación.

4.1.1 Introducción

Gao, Tsao y Wu [GTW04] definen la automatización de las pruebas de *software* como aquellas actividades y esfuerzos realizados con la intención de automatizar tareas de ingeniería y operaciones en un proceso de pruebas utilizando estrategias bien definidas y soluciones sistemáticas. Es preciso contar para esto con una infraestructura que como mínimo permita ocultar las complejidades brindando a sus usuarios la posibilidad de elaboración, mantenimiento y ejecución de las pruebas de forma sencilla [Mes07]; así como la obtención y registro de resultados, mantener la documentación, trazabilidad y el seguimiento de incidentes.

Automatizar puede significativamente reducir el esfuerzo requerido para efectuar las pruebas en forma adecuada, de formas más eficientes, o significativamente incrementar la cantidad de pruebas que pueden llevarse a cabo en un lapso determinado. Se han logrado ahorros de hasta un 80% del esfuerzo manual. Se ha ahorrado en los costos o generado productos de mejor calidad y más rápidamente [FG99]. Sin embargo, las ventajas de la automatización en general son alcanzables a mediano y largo plazo, dependiendo de cada tipo de herramienta, no estando garantizado el éxito solo por su mera utilización [ISQ07].

En algunas casos puede resultar imprescindible apoyarse en dicha infraestructura, ya sea debido a las características de las pruebas a realizar (por ejemplo en pruebas de carga, de desempeño, de regresión, de cobertura de código, de configuraciones) u oficiando de soporte a metodologías de diseño basadas en TDD (3.3), en cuyo caso la verificación automática del comportamiento del *software* puede ser vista también como uno de los mayores avances en los métodos de desarrollo en las décadas recientes [Mes07].

4.1.2 Objetivos

La automatización de las pruebas debería establecerse en función del cumplimiento de objetivos cuyo grado de satisfacción sea cuantificable. Los atributos de calidad deben ser medidos para poder ser comparados y establecer, de esa forma, el grado de cumplimiento de los objetivos fijados teniendo en cuenta las restricciones que puedan aplicar en cada caso. Fewster y Graham sugieren elegir unos pocos atributos principales que brinden la mayor información acerca de si se alcanzaron o no los objetivos propuestos [FG99].

Entre los posibles atributos generales pueden citarse los siguientes: eficiencia, facilidad de inserción, fiabilidad, flexibilidad, facilidad de mantenimiento, portabilidad, robustez y usabilidad. Cada organización o equipo debería comenzar por determinar los objetivos a cumplirse con la automatización, analizando para cada proyecto las particularidades del mismo a los efectos de utilizar la mejor estrategia y herramientas de automatización que pueda disponer para cada caso [FG99].

4.1.3 Clasificación de las herramientas

Existe una gran cantidad de herramientas para automatizar diferentes aspectos relacionados con las actividades de V&V, a lo largo del ciclo de vida. A continuación se exhibe una clasificación posible [ISQ07] que contempla no solo las herramientas directamente involucradas en la prueba (dinámica) sino también otras herramientas para la gestión del proceso así como para las técnicas estáticas:

- Administración de las pruebas y del proceso de pruebas:
    - Administración de las pruebas.
    - Seguimiento de incidentes.
    - Gestión de la configuración.
    - Administración de requerimientos y trazabilidad desde las pruebas.
- Enfoque estático:
    - Apoyo al proceso de revisión.
    - Análisis estático.
    - Modelado.
- Especificación de las pruebas:
    - Diseño de las pruebas.
    - Preparación de datos de prueba.
- Ejecución de las pruebas y registro:
    - Ejecución de casos de prueba.
    - Marcos (*frameworks*).
    - Comparadores.
    - Medición del cubrimiento.
    - Seguridad.
- Desempeño y monitorización:
    - Análisis dinámico.
    - Desempeño, carga y estrés.
    - Monitorización.
- Específicas
    - Especializada para un tipo de aplicación.
    - Especializada para un área de aplicación.

4.1.4 Estrategias.

Existe una amplia variedad de herramientas en el mercado. Se pueden reconocer dos grandes estrategias para la automatización de la ejecución: “Captura y

Reproducción" por un lado y "Guiones" por otro. La primera implica el uso de herramientas para grabar la interacción con el SUT mientras es probado manualmente, la segunda involucra la escritura de guiones (*scripts*) para ejercitar el SUT. La frontera entre ambas, sin embargo, es difusa, muchas de las herramientas de este último tipo (guiones) proveen también de facilidades de grabación [Mes07].

Captura y reproducción

Las técnicas de captura y reproducción permiten al verificador capturar y grabar pruebas. Posteriormente puede editarlas, modificarlas y reproducirlas en distintos entornos. Dentro de este tipo de herramientas, aquellas que graban la interfase de usuario a nivel de componentes (y no de *bitmaps*) son más útiles. Durante la grabación se capturan las acciones realizadas por el verificador, creando automáticamente un guión (*script*) en algún lenguaje de alto nivel. Luego el verificador puede modificar el guión para crear una prueba reutilizable y viable de mantener. El guión se vuelve la línea base. Luego es reproducido en una nueva versión, contra la cual es comparado. Estas herramientas vienen en general acompañadas de un comparador, que compara automáticamente la salida en el momento de ejecutar el guión con la salida grabada, registrando los resultados [Dus03]. Ciertos autores consideran que la simple captura manual de la sesión y su reproducción posterior no aportan realmente una "verdadera" automatización [FG99] [KBP01].

Guiones

Dentro de esta estrategia se pueden establecer diferentes niveles [FG99]:

- Lineal: Sin sentencias de control de flujo. Generalmente son de este tipo los guiones obtenidos directamente con los programas de captura.
- Estructurados: Los guiones contienen sentencias de control de flujo.
- Compartidos: Se definen guiones comunes a varias pruebas.
- Impulsados por datos (*data-driven*): Las entradas para la prueba residen en archivos de datos, separados del guión. Esto permite que el mismo guión sea usado para diferentes pruebas, se reduce la duplicación e independiza el guión de los datos, lo que permite entregar los archivos a los expertos del negocio para que los pueblen con datos. Con esta técnica comienzan los verdaderos beneficios.
- Impulsados por comandos (*keyword-driven*): Es una extensión de los impulsados por datos, donde una porción de la lógica se desplaza hacia los archivos de datos donde reside la entrada; de esta manera además de datos se especifican tareas mediante comandos, que serán interpretados por guiones apropiados.

Las dimensiones de las alternativas posibles pueden ser expuestas en una matriz como la que muestra la figura 4-1, estableciéndose un criterio de selección [Mes07].

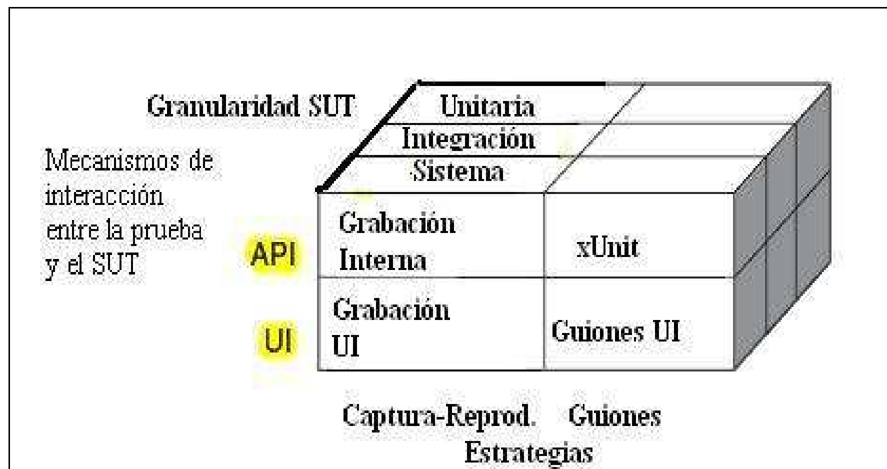


Fig. 4- 1: Matriz de selección. Adaptada de [Mes07].

4.1.5 Consideraciones

Entre los múltiples beneficios de la automatización de las pruebas se mencionarán los siguientes, tomando como fuentes a [BA04], [FG99], [ISQ07], [KBP01], [Mes07]: reducir el trabajo repetitivo, mayor consistencia y cobertura, evaluación objetiva, facilidad para el acceso a la información de las pruebas, ejecución más habitual, ejecución de pruebas imposibles de ejecutar en forma manual, reutilización, escalabilidad, brindar confianza, soporte de TDD y al desarrollo ágil, obtención de métricas.

Más allá de los beneficios enumerados precedentemente, muchas de las tecnologías existentes han sido consideradas primitivas o carentes de la calidad deseada, ayudando a que una infraestructura inadecuada para pruebas provoque pérdidas económicas enormes [Tas02]. Resulta también inquietante la apreciación de Fewster y Graham acerca que entre un 40% y un 50% de las incorporaciones de herramientas para ejecutar pruebas en forma automatizada habían caído en desuso o jamás habían sido utilizadas [FG99]. Muchas de las tecnologías para pruebas de programas tradicionalmente han resultado costosas al momento de su adquisición, dificultosas de poner en práctica y consumidoras luego de gran cantidad de recursos [LW04].

Entre los principales riesgos se mencionarán los siguientes [FG99] [GTW04] [ISQ07] [KBP01]:

- **Las expectativas poco realistas.**
- **Subestimar:**
  - El tiempo, costo y esfuerzo de su introducción.
  - El tiempo y el esfuerzo necesarios para alcanzar ventajas significativas y continuas.
  - El esfuerzo requerido para mantener los elementos de prueba.
- **El intento de sustitución de todas las pruebas manuales.**
- **La ausencia de:**

    - Estrategia y procesos bien definidos.
    - Recursos apropiados.
    - Formación adecuada.
    - Asignación correcta de roles y responsabilidades.
    - Esquema de selección y evaluación de herramientas y procesos.
- **Incorporar la automatización en forma tardía** a los proyectos de desarrollo.
- **Falsa sensación de seguridad.**

En cuanto a las limitaciones [FG99] [HT99] [KBP01]:

- **Puede no ser conveniente automatizar ciertas pruebas**, por ejemplo: interfases de usuario volátiles, pruebas cuya ejecución es esporádica, pruebas con resultados fácilmente verificables por humanos o dificultosamente verificables por computador.
- **En ciertas pruebas debe corresponder a un humano la verificación**, como ser en las de "usabilidad".
- **Las pruebas manuales y las técnicas estáticas logran encontrar más defectos que las pruebas automatizadas**. Las técnicas estáticas manuales (inspecciones, recorridas) y las pruebas exploratorias manuales deben ocupar un sitial importante dentro del proceso general de pruebas. Las pruebas de regresión en general no encontrarán nuevos errores, más allá de su importancia estratégica en apoyo de la reconstrucción de código, el desarrollo incremental y la integración (*build*).
- **La automatización no hará necesariamente más efectivas a las mismas pruebas ejecutadas manualmente** (de ser posible hacerlo).
- **La automatización de las pruebas funcionales o de validación** presenta enormes desafíos, especialmente cuando la visibilidad se obtiene principalmente mediante interfases GUI, si bien se han efectuado avances importantes, en especial para la prueba funcional automatizada de aplicaciones *web*. Es necesaria una nueva generación de herramientas [And07].
- **El problema del oráculo.** (2.1.8) Contar con un oráculo correcto y completo surge como uno de los más importantes y complejos problemas a resolver.

Si bien la carga de la culpa por los fracasos fácilmente puede recaer en las herramientas, también deben buscarse las razones en la cantidad de recursos puestos en educación, en la forma en que se introducen dichas herramientas en las organizaciones, en el tipo de pruebas que permiten realizar y las limitaciones que tienen, en los procesos utilizados, en las particularidades de las aplicaciones a probar y la infraestructura sobre las que se apoyan, así como en la pretensión de automatizar todas las pruebas o las expectativas, rápidamente frustradas, acerca que se reducirán tiempos y costos relacionados con todas las pruebas al automatizarlas.

Es de destacar la gran importancia de construir las aplicaciones teniendo en cuenta los aspectos que faciliten la verificación (*testability*[26]), como ser una correcta y precisa separación en capas, "ortogonalidad" y desacoplamiento [Bec02] [HT99].

Resulta vital comprender que los procesos de automatización implican procesos de desarrollo, la organización debe tomar conciencia que se está desarrollando *software* y que debe aplicarse una metodología adecuada al respecto [KBP01]. Tampoco debe despreciarse el problema sobre la confiabilidad de las pruebas; ya que la programación y diseño de las mismas está sujeta a errores y debería someterse a alguna forma de verificación.

Deben abordarse además ciertas cuestiones claves al seleccionar las herramientas a utilizar, a saber: escalabilidad, reutilización, versionado, independencia de plataforma, costo, vendedor, adaptabilidad, etc. En los últimos años parece haberse consolidado una tendencia hacia desarrollar y adaptar herramientas por parte de los propios desarrolladores y verificadores, muchas de las cuales han sido liberadas bajo licencias de código libre. La posibilidad de contar con el código para adaptarlas, su construcción colectiva, la sinergia que se crea a su alrededor, el costo del licenciamiento de las herramientas comerciales, la posibilidad de probarlas enteramente sin incurrir en tales costos, la escritura de código en el mismo lenguaje que la aplicación, la necesaria variedad que adapte al contexto y la necesidad creciente de herramientas de automatización para soportar las prácticas de metodologías ágiles, cuenten muy probablemente entre las múltiples razones de su rápida popularidad y aceptación [Pet104] [Pet204]. Resulta notoria la creciente prominencia de este tipo de herramientas *open source* en general en la Ingeniería de *Software*, especialmente en soporte de las metodologías ágiles y en particular de XP. Una muy amplia lista de estas herramientas *open source* está publicada en [OST07].

4.2 Tecnología xUnit para pruebas unitarias

Se estudiará acerca de xUnit, un modelo de pruebas y conjunto de marcos para pruebas unitarias. Se ofrecerá una reseña, descripción de su arquitectura, ámbitos de aplicación, ejemplos y referencias a sus múltiples extensiones.

4.2.1 Introducción

Kent Beck en su ensayo "*Simple SmallTalk Testing: With Patterns*" [Bec94] propuso una estrategia de pruebas y un marco para pruebas unitarias comandado por código. La filosofía de esa propuesta consistió en crear y ejecutar pruebas embebidas por el marco, escribiéndolas y controlando los resultados utilizando el mismo lenguaje en que está escrito el programa a probar. Instó a los desarrolladores a escribir sus propias pruebas e invertir en ello no menos de un cuarto y hasta la mitad del tiempo empleado para la elaboración de sus programas.

[26] Grado en que un sistema o componente facilita el establecimiento de criterios de prueba y el rendimiento de dichas pruebas para determinar si se cumplieron los criterios [IEEE Std 610.12-1990].

Sostuvo que las pruebas deben aislarse y su ambiente ser creado y restaurado antes y después de su ejecución, postulando los siguientes patrones básicos:

- *Fixture*[27]: Refiere a crear e inicializar los objetos necesarios para la prueba denominándose como tal a una configuración simple cuyo comportamiento es predecible.
- Caso de prueba (*Test Case*): Refiere a la acción de agregar métodos a la clase de prueba para estimular la "*fixture*", constituyéndose en simples unidades de prueba.
- Comprobación (*Check*): Refiere a la verificación contra resultados esperados. El marco verificará cada comprobación, ante una falla detendrá la ejecución del método de prueba en curso y comenzará con el siguiente.
- Conjunto de pruebas (*Test Suite*): Los casos de prueba se agrupan en objetos que representan un conjunto de casos de prueba.

La aplicación de los patrones identificados se tradujo en implementar porciones de código que estimularan objetos para ejecutar los casos de prueba, individualmente o agrupados en conjuntos, pudiéndose comprobar los resultados contra aquellos esperados, produciéndose un ritmo según la siguiente estructura recurrente [RS05]: "Crear un objeto, Estimularle, Comprobar el resultado". Este patrón resultante suele ser descrito por la sigla "AAA"[28] (*Arrange, Act, Assert*).

Bajo la óptica del modelo xUnit una prueba unitaria es una pequeña e incremental prueba que los programadores aplican a una simple unidad de trabajo en aislación del resto, utilizándose en general para su programación el mismo lenguaje en que está programado el SUT.

Kent Beck [Bec02] advierte que al repetirse el patrón a diferentes escalas la cuestión es acerca de cuántas veces habrá que crear objetos para probarles, lo que provoca que dos restricciones entren en conflicto: rendimiento y asilamiento. Para lograr ejecutar eficientemente se deberán crear los objetos comunes una única vez, mientras que el suceso o falla de una prueba debe ser irrelevante para las demás. Así mismo, los recursos adquiridos durante la ejecución deben ser liberados. La figura 4.2 ilustra este ciclo.

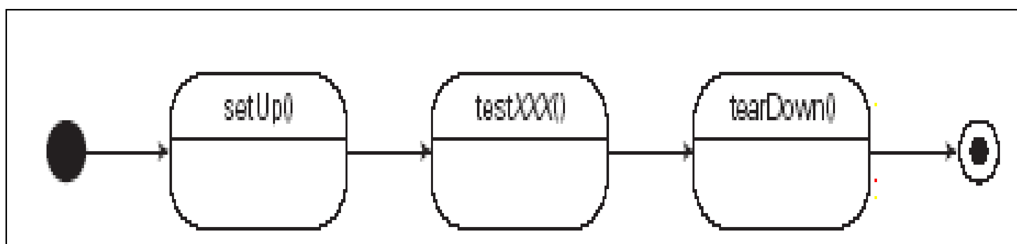


Fig. 4- 2: Ciclo de vida de *TestCase* [MH04 fig.2.6].

[27] Se opta por no utilizar traducción del término "*fixture*" en este ámbito.
[28] Acuñada por Bill Wake, según se cita en [RS05] y en [Bec02].

Según Meszaros [Mes07] la mayoría de los xUnit han sido implementados usando un lenguaje orientado a objetos (OOPL) y tienen en común el proveer un conjunto básico de funcionalidades que siguen los patrones expuestos por Beck [Bec94] [Bec02], permitiendo:

- Definir una prueba[29]: Cada prueba se representa por un método de prueba que es identificado por medio de marcadores o convenciones de nombres. Se implementa una prueba de cuatro fases siguiendo un patrón que denomina "*Four Phase Test*": Inicializa la prueba, ejercita el SUT, comprueba los resultados, libera recursos. Estos métodos de prueba pueden pertenecer a clases de prueba (*Test Clases*).
- Comprobar[30]: Se especifican los resultados esperados con invocaciones a métodos de comprobación (*Assert Methods*). Estos métodos implementan el oráculo que queda incluido en el código de la prueba. Según Beck [Bec02] las decisiones deben ser "booleanas" y comprobables por el computador. También es posible comprobar excepciones esperadas[31], en cuyo caso la falla es notificada solo si la excepción no se produce.
- Ejecutar: Estos marcos ofrecen varias formas de comandar la ejecución de las pruebas y obtener los resultados. Proveen de un módulo encargado de esta tarea, pudiéndose utilizar desde la línea de comandos, bajo una interfase gráfica propia o integrado a una herramienta de desarrollo.
- Agrupar[32]: Las pruebas se pueden agrupar en conjuntos, de ese modo pueden ser ejecutadas con un única operación (*Suites*).
- Exponer resultados: La razón primera de ejecutar pruebas es conocer los resultados. Estos deben ser percibidos de manera no ambigua y fácilmente inteligible. Se sigue la premisa que si "no hay novedades, entonces son buenas noticias" ("*No news, good news*"). La prueba se detiene ante la falla, aunque no impide seguir ejecutando el resto de las pruebas. Se utiliza una semántica de colores definida por: verde= pasa, rojo = falla.

Finalmente, se destaca que el marco de pruebas unitarias propuesto por Beck ha sido extendido a más de 30 lenguajes [Bec02] y su versión para Java, denominada Junit [Jun107], herramienta liberada bajo licencia CPL[33], se ha transformado en el estándar de hecho para desarrollar pruebas unitarias en Java [MH04]. Actualmente dichos marcos se incorporan a IDEs de desarrollo y se suministran para los mismos gran cantidad de extensiones [OST07].

[29] Corresponde al patrón "*Test Method*" identificado por Kent Beck en [Bec02].
[30] Corresponde al patrón "*Assertion*" identificado por Kent Beck en [Bec02].
[31] Corresponde al patrón *"Exception Expected*" identificado por Kent Beck en [Bec02].
[32] Corresponde al patrón "*AllTest*" identificado por Kent Beck en [Bec02].
[33] *Common Public Licence*. http://www.opensource.org/licenses/cpl1.0.php.

4.2.2 Escritura de las pruebas

En este apartado se brindarán ejemplos concretos acerca de cómo se codifican las pruebas utilizando JUnit. Los ejemplos suministrados corresponden al lenguaje Java [Jav07] y a JUnit 4.0, el cual introdujo una API[34] diferente de sus versiones anteriores. La sintaxis y las posibilidades que brinda se mostrarán exponiendo fragmentos de código. En cuanto a los demás xUnit para lenguajes orientados a objetos la escritura de las pruebas sigue reglas básicas similares, con variaciones dependientes de los lenguajes de implementación o sabores del marco específico.

El primer ejemplo se presenta en las figuras 4-3 a 4-5, correspondientes al código de un caso de prueba que verifica si determinado libro está en el catálogo de una biblioteca. Se puede observar en dicho fragmento de código lo siguiente:

- El método *BookAvailabilityinLibrary* está precedido de la anotación @Test, lo cual indica que se trata de un método de prueba. Dicho método se ha programado para verificar si el método del SUT denominado *ChekAvailabilitybyTitle* encuentra el texto buscado.
- El resultado de la prueba será validado con el método *AssertEqual*, que implementa el oráculo, comparando entre el resultado de la búsqueda y el resultado esperado.
- Los métodos precedidos de las anotaciones @Before y @After serán ejecutados en forma automática respectivamente antes y después de la prueba en tanto que @BeforeClass y @AfterClass informan al marco que los métodos antecedidos de esa forma se deben ejecutar una única vez cada uno, al comienzo y final del conjunto de pruebas contenido en una clase.
- El manejo de excepciones se logra mediante la utilización de un parámetro en los métodos con anotación @Test. En el caso del ejemplo el método de prueba ya mencionado pasa solo si *BookNotAvailableException* es lanzada; y falla si la excepción no es lanzada o si se dispara otra diferente [Dos05].

Existen otros parámetros, anotaciones y múltiples métodos de Assert además de los ejemplificados. A modo de ejemplo se mencionará que es posible especificar el tiempo máximo de tolerancia pasado el cual un método falla (mediante el parámetro *timeout=*) y se ofrece una anotación, @Ignore, que permite evitar disparar el método así anotado durante la ejecución.

[34] *Application Programming Interfase.*

```
package example.junit4;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;

public class LibraryTestUsingSetup{

    private Library library;

    @Before public void runBeforeEachTest(){
        library = new Library();
    }

    @Test public void bookAvailableInLibrary(){
        boolean result = library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                true,
                result);
    }

    @After public void runAfterEachTest(){
        library = null;
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTestUsingSetup.class);
    }
}
```

Importa desde el marco

Se ejecuta previo a cada prueba

Verificación

Se ejecuta luego de cada prueba

Fig. 4- 3: Codificación en JUnit [Dos05].

```
package example.junit4;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import junit.framework.JUnit4TestAdapter;

public class LibraryTestUsingBeforeClass{
    private Library library;
    @BeforeClass public void runOnceBeforeAllTests(){
        library = Library.newInstance(10020);
        library.connectToCentralLibrary();
        library.synchronizeDataWithCentralLibrary();
    }

    @Test public void bookNotAvailableInLibrary(){
        assertFalse("Our Library should not have this old book",
                library.checkAvailabilityByTitle("Really Old Book"));
    }

    @Test public void bookAvailableInCentralLibrary(){
        assertTrue("Central Library should have this book",
                library.checkAvailabilityInCentralLibrary("Really Old Book"));
    }

    @AfterClass public void runAfterAllTests(){
        library.disconnectFromCentralLibrary();
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTestUsingBeforeClass.class);
    }
```

Se ejecuta previo a todas las pruebas.

Se ejecuta luego de todas las pruebas.

Fig. 4- 4: Codificación en JUnit [Dos05].

```
@Test(expected=BookNotAvailableException.class)
public void bookNotAvailableInLibrary(){
    Library library = new Library();
    library.checkAvailabilityByTitle("Some book that does not exist");
}
```

Fig. 4- 5: Esperando una excepción en JUnit [Dos05].

Es de destacar que la versión 4.4 introduce más novedades [Jun207] [Har05]:

- Nueva sintaxis para los métodos *Assert*, recogiendo el aporte de Joe Walnes [Wal05] que permite indicar condiciones más naturalmente, combinándolas o negándolas, por ejemplo: "*assertThat(x, is(not(4)))*".
- Un conjunto de predicados (*matchers*) para utilizar con *assertThat*; a partir de la inclusión de la biblioteca del "Proyecto Hamcrest"[35] [HaC07].
- Asunción: Permite expresar una restricción.
- Una nueva clase de declaraciones de intención llamadas "*Theories*" (Teorías), absorbidas desde el "Proyecto Popper" [Pop07], que expresan una declaración sobre el comportamiento del código que puede ser cierta sobre infinitos conjuntos de valores de entrada. Estas surgieron, según Saff [Saf07], de la motivación provocada por la posibilidad de no detectar fallas no importando cuan representativos sean los escenarios planteados: una combinación de datos de entrada no prevista en los ejemplos concretos podría hacer fallar la unidad en una ejecución futura. Saff y Boshernitsan [SB07] propusieron entonces un mecanismo para mitigar este riesgo, consistente en el suministro de especificaciones parciales sobre un conjunto de valores infinito o muy grande, como extensión a la disciplina TDD. Estas Teorías son universalmente cuantificables. Todas las confirmaciones (Assert) deben mantenerse para cualquier combinación posible de valores que pasan lo asumido. Se permite de esta forma al desarrollador establecer y verificar propiedades generales. Las figuras 4-6 y 4-7 brindan ejemplos comentados de su codificación.

```
@RunWith(Theories.class)
  public class UserTest {
  @DataPoint public static String GOOD_USERNAME = "optimus";
  @DataPoint public static String USERNAME_WITH_SLASH =  "optimus/prime";

@Theory public void filenameIncludesUsername(String username) {
      assumeThat(username, not(containsString("/")));
      assertThat(new User(username).configFileName(),  containsString(username));
  }
}
```


Fig. 4- 6: *Theoris* en JUnit [Jun207].

[35] Es la primera vez que JUnit incorpora directamente clases de terceros [Jun207].

Al ejecutarse una teoría se consideran todos los valores anotados con @DataPoint. Cuando una teoría recibe argumentos que no cumplen con la asunción establecida por el método *assumeThat* se termina con su ejecución. Si ninguno de los @DataPoint satisface lo asumido por la teoría, entonces se marcará como que falló. Se provee de esa forma soporte para explorar nuevos @DataPoint que violen la teoría. Cada entrada fallida deberá ser analizada por el desarrollador para reconocer si es una falla o si falta una asunción.

```
// Definición de la teoría

@Theory defnOfSquareRoot(double n) {
assumeTrue(n >= 0);
assertEquals(n, sqrRoot(n) * sqrRoot(n), 0.01);
assertTrue(sqrRoot(n) >= 0);
}
// Se proveen diversas entradas para su evaluación en ejecución
@RunWith(Theories) public class SquareRoot {
        @DataPoint public double FOUR = 4.0;
        @DataPoint public double NINE = 9.0;
        @Theory defnOfSquareRoot(double n) { ... }
        }
```

Asume n >= 0

Comprueba de forma genérica.

Fig. 4- 7: *Theorys* en JUnit [Saf07].

Se itera sobre la ejecución de las teorías hasta que no se encuentran más entradas que violen las mismas. Es posible someter el código a análisis automatizado para inferir conjuntos de @DataPoint para su prueba [Saf07].

- Definición y ejecución de pruebas parametrizadas, en las cuales se ejecuta una vez para cada grupo de parámetros y valores esperados suministrados bajo las anotaciones @Parameters y @Parameter (ver figura 4-8).

```
import org.junit.*;
import static org.junit.Assert.*;
public class AdditionTest {
 @Parameters public static int[][] data
 = new int[][] {
  {0, 0, 0},
  {4, 3, 1},
  {5, 5, 0},
  {6, 8, -2}
  };
 @Parameter(0) public int expected;
 @Parameter(1) public int input1;
 @Parameter(2) public int input2;
@Test public void testAddition() {
assertEquals(expected, add(input1, input2));
 }
private int add(int m, int n) {
return m+n;
 }
}
```

Cada terna contiene el valor esperado y los valores de entrada.

Fig. 4- 8: Pruebas parametrizadas en JUnit [Har05].

Escritura automática de las pruebas

Existe un conjunto de herramientas y extensiones de estos marcos, surgidas a partir de varias líneas de investigación, las cuales posibilitan la creación automatizada de casos de prueba unitarios. Para lograrlo infieren a partir de generalizaciones (como ser teorías o pruebas parametrizadas), de pre o post condiciones del contrato, de diagramas de estado para pruebas de comportamiento y de la inspección del código en el caso de ausencia de especificaciones. Se pueden referenciar al respecto el proyecto MUTT [MUT07] [TSG05] [TS06] y su continuación, el proyecto PEX [PEX07], ambos para .Net [Net07]. También *Theory Explorer* [SB07] [Saf07], *JUnit Factory* [JuF07] y el proyecto *JFing* [JFi05] [JFi06].

4.2.3 Ejecución de las pruebas

Los marcos ofrecen programas para ejecutar las pruebas desde la línea de comandos o desde una interfase gráfica. En la figura 4-9 se muestra una imagen de la pantalla de ejecución de NUnit [Nun07] (marco xUnit para .Net). JUnit 4 no brinda interfase gráfica pero es posible incorporarle a otras herramientas o IDEs, por ejemplo a Eclipse [Ecl07] [Ame07].

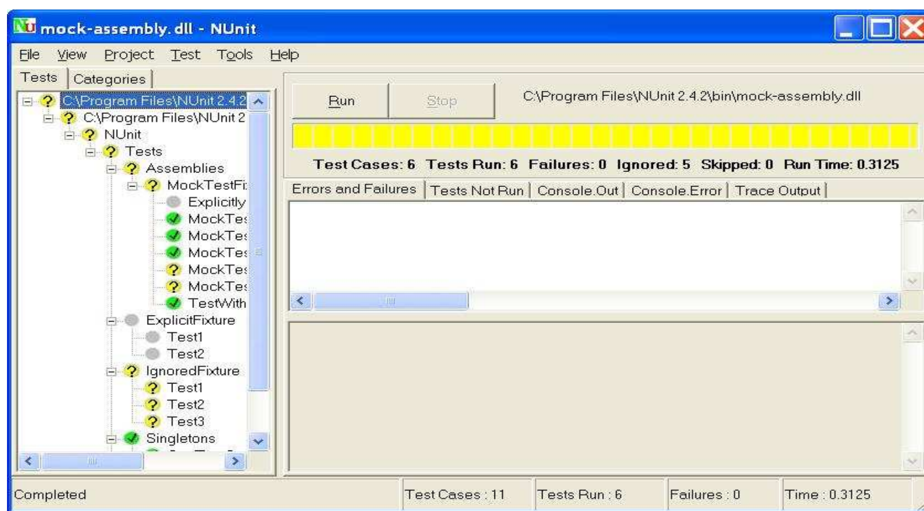


Fig. 4- 9: NUnit.

4.2.4 Terminología

Según Kent Beck el término *fixture* (4.2.1) [Bec94] refiere a una configuración simple con comportamiento predecible, por ejemplo un programa que realiza una conexión de red hacia una máquina conocida, donde la predicción es que responderá a los paquetes de red; mientras que "*test case*" es el estímulo para provocar la situación predecible que se verificará. Desde el punto de vista del marco las pruebas están representadas por un objeto del programador denominado *TestCase*, en tanto *TestSuite* es un conjunto de objetos *TestCase* o *TestSuite*.

Massol & Husted [MH04] definen, en el marco JUnit, a una *fixture* como el conjunto de recursos comunes o datos necesarios para ejecutar una o más pruebas, en tanto que *TestCase* es una clase que ejercita comportamiento mediante métodos para prueba que denominan *tests*. Cada *fixture* será automáticamente creada y eliminada por un *TestCase.* Indican que un caso típico podría estar representado por una conexión a una base de datos necesaria para la ejecución de varias pruebas, donde una *fixture* común sería la encargada de hacerla, evitando la duplicación de código. Un *TestCase* agrupará varios *test* que podrán compartir una misma *fixture*.

Según Rainsberger y Stirling [RS05] una *fixture* es una forma natural que guía la agrupación de pruebas: deben identificarse las pruebas que comparten *fixtures* para factorizarlas en una misma clase. Establecen que siguiendo la definición de Beck [Bec94] la primera parte de una prueba (creación de objetos) puede ser mencionada como creación de *fixtures*. Indican que no deben confundirse los términos "*test case*" y la clase "*TestCase*": el primero refiere a una prueba simple que verifica un camino dentro del código, implementada como un método.

Por último, Meszaros [Mes07] resume como "*test fixture",* para el conjunto de los marcos xUnit, como todo lo necesario que debe codificarse para preparar la prueba y esperar un resultado, incluyendo típicamente una instancia de la clase cuyos métodos se están probando y otros objetos de los cuales el SUT depende. Refiere entonces a las precondiciones de la prueba. Advierte que varios miembros de la familia xUnit (en especial NUnit) denominan a la clase "*TestCase"* como "*test fixture*".

Se ofrece a continuación una tabla comparativa de terminología entre diferentes marcos xUnit basada en la tabla de referencias cruzadas brindada en [Mes07].

					Nivel Fixture		Nivel conjunto de pruebas		
lenguaje	Miembro xUnit	Clase TestCase	Test Suite Conjunto de pruebas	Método de prueba	Fixture setup Inicialización	Fixture teardown Restauración	Suite Fixture Setup	Suite Fixture Teardown	Expected Exception
Java 1.4	JUnit 3.8.2	Subclass of TestCase	static suite()	testXxx()	setUp()	tearDown()	Not applicable	Not applicable	Subclass of Expected Exception Test
Java 5	JUnit 4.0+	import org. junit.Test	static suite()	@Test	@Before	@After	@Before Class	@After Class	@Exception
.NET	CsUnit	[TestFixture]	[Suite]	[Test]	[SetUp]	[TearDown]	Not applicable	Not applicable	[Expected Exception()]
.NET	NUnit 2.0	[TestFixture]	[Suite]	[Test]	[SetUp]	[TearDown]	Not applicable	Not applicable	[Expected Exception()]
.NET	NUnit 2.1+	[TestFixture]	[Suite]	[Test]	[SetUp]	[TearDown]	[Test Fixture SetUp]	[TestFixture TearDown]	[Expected Exception()]
.NET	MbUnit 2.0	[TestFixture]	[Suite]	[Test]	[SetUp]	[TearDown]	[Fixture Setup]	[Fixture Teardown]	[Expected Exception()]
.NET	MSTest	[TestClass]	Not applicable	[Test Method]	[Test Initialize]	[Test Cleanup]	[Class Initialize]	[Class Cleanup]	[Expected Exception()]
PHP	PHPUnit	Subclass of TestCase	static suite()	testXxx()	setUp()	tearDown()	Not applicable	Not applicable	Subclass of Expected Exception Test
Python	PyUnit	Subclass of unittest. TestCase	Test Loader()	testXxx	setUp	tearDown	Not applicable	Not applicable	assert raise
Ruby	Test::Unit	Subclass of Test::Unit:: TestCase	Classname. suite()	testXxx()	setup()	teardown	Not applicable	Not applicable	assert_raise
Smalltalk	SUnit	Superclass: TestCase	TestSuite named:	testXxx	setUp	tearDown	To be determined	To be determined	should:raise:
VB 6	VbUnit	Implements IFixture	Implements ISuite	TestXxx()	IFixture_ Setup()	IFixture_ TearDown	IFixture Frame_ Create()	IFixture Frame_ Destroy	on error...
SAP ABAP	ABAP Unit	FOR TESTING	Automatic	Any	setup	teardown	class_setup	class_ teardown	To be determined

Tabla 4- 1: Comparativo de la terminología xUnit [basada en Mes07 pp.745-746].

4.2.5 Arquitectura

Según Hamill [Ham04], todos los xUnit (OOLP) tienen una arquitectura similar. La figura 4-10, extraída de la misma fuente, muestra el diagrama de clases del núcleo para todo marco de este tipo; en tanto la figura 4-11 expone el correspondiente a JUnit, en su versión 3.81, suministrado por Kent Beck y Erich Gamma [BG99] quienes identificaron, adicionalmente, los patrones de diseño utilizados.

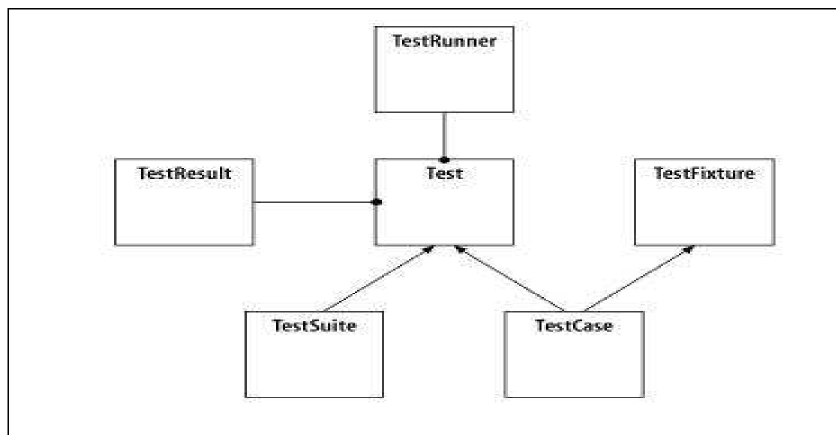


Fig. 4- 10: Diagrama básico de clases xUnit [HAM04 fig. 3-6].

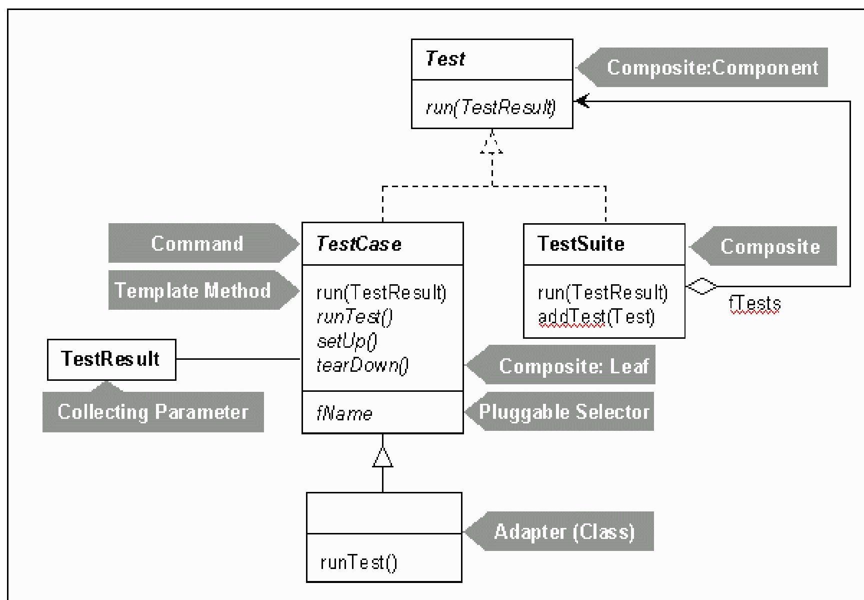


Fig. 4- 11: Patrones de diseño aplicados a JUnit [BG99].

Meszaros [Mes07] explica que en el momento de la ejecución existe una diferencia fundamental para el caso de NUnit (versión 2.x), dado que en lugar de "instanciar" un objeto "comando" para representar cada método de prueba a ejecutar, crea una única instancia (que denominan *test fixture*) a reutilizar por cada método.

Extensiones

Existe una gran cantidad y variedad de extensiones y herramientas que pueden asociarse a los marcos xUnit. Entre otros en los sitios [Jun107], [OST07] y [Nun07] es posible encontrar una lista y vínculos acerca de extensiones y herramientas relacionadas a estos marcos.

4.3 Dobles para pruebas

En esta sección se tratará de las técnicas consistentes en la sustitución momentánea de porciones de código por otras, a los efectos de la ejecución de pruebas unitarias. Se explicará sobre los diferentes tipos de unidades sustitutivas, las técnicas más comunes y su vinculación con TDD.

4.3.1 Introducción

La sustitución momentánea de porciones de código (objetos, componentes, módulos, etc.) por otros que no implementan la funcionalidad completa requerida, o que no serán utilizados en producción, ha sido una práctica habitual tanto para las pruebas como para el desarrollo.

La combinación de los diferentes módulos que constituyen un programa o sistema es conocida como integración. Tal como se trató en (2.3.2), entre las diversas estrategias para esta integración se pueden utilizar aquellas basadas en una estrategia incremental o guiadas por la arquitectura según los hilos funcionales previamente identificados. Para llevar a cabo estas estrategias es necesario utilizar módulos provisorios, conocidos en general como *stubs* (sustituyen a los módulos invocados) y *drivers* (sustituyen a los módulos que efectúan la invocación), para reemplazar momentáneamente a los reales o para disparar y controlar su ejecución, aunque puede establecerse una clasificación con mayor granularidad, por lo menos en OOPL, tal como se mencionará más adelante (4.3.2).

En el ámbito de las pruebas unitarias se utilizan este tipo de reemplazos, tanto para comandar la prueba como para lograr que se desarrolle sin interferencias por parte de las unidades reales de los cuales la unidad a probar depende o cuando estas unidades no están disponibles [Bei90].

Meszaros [Mes07] se refiere a los componentes sustitutivos de otros como “dobles para pruebas”. Estos reemplazos brindan, durante la prueba, entradas indirectas al SUT (no son las entradas directamente proporcionadas por la prueba a través de la interfase principal) o sirven para verificar salidas indirectas, especialmente referidas a la comunicación con los componentes “colaboradores”.

Por último, si bien los ejemplos y referencias que se brindarán en el resto de la sección son relativos a OOLP, no necesariamente todas las técnicas y reemplazos que se mencionarán son aplicables solo en dicho ámbito.

4.3.2 Clasificación

Se adoptará la clasificación propuesta por Meszaros [Mes07], la cual distingue los siguientes tipos de "dobles para pruebas":

- ***Dummys:*** Son objetos pasados como parámetros pero nunca utilizados.
- ***Fakes:*** Son objetos que actualmente corresponden a una implementación que funciona, pero que no serán utilizados en producción, como ser una base de datos en memoria. Al igual que los anteriores, no se necesita configurar respuestas o expectativas, simplemente se sustituye el objeto real por su reemplazo y se utiliza.
- ***Stubs**:* Son objetos que responden con un conjunto de respuestas premeditadas a las llamadas que se les efectúan Se les define con una implementación orientada hacia la prueba a realizar, de la interfase del objeto del cual el SUT depende. Los *stubs* se han utilizado históricamente de forma habitual durante las pruebas, en especial cuando se desea sustituir un módulo con el cual el objeto a probar debe comunicarse pero que no está disponible, su implementación es compleja o tiene efectos no deseados durante el desarrollo de la prueba [Bei90] [Mes07]. En la figura 4-12 se expone un fragmento de código correspondiente a la utilización de un *stub* extraída de [Fow207], donde lo que se sustituye es un servicio de envío de correo electrónico por un "doble" que implementa la interfase.

```
public interface MailService {
public void send (Message msg);
}
public class MailServiceStub implements MailService {
private List<Message> messages = new ArrayList<Message>();
public void send (Message msg) {
messages.add(msg);
}
public int numberSent() {
return messages.size();
}
}

class OrderStateTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent());
  }
```

Fig. 4- 12: Código y uso de un *Stub* [Fow207].

Los *stubs* actúan como puntos de control brindando entradas indirectas al SUT. Según sean estas entradas, Meszaros establece una clasificación que incluye a los que denomina "*responders*" (respondedores), aquellos que brindan entradas tanto válidas como no válidas vía un retorno normal, "*saboteur*s" (saboteadores), a los que inyectan entradas anormales para provocar excepciones y "*temporaries*" (temporarios), aquellos que reemplazan colaboradores aún no disponibles o inexistentes pero necesarios. La figura siguiente extraída de [Mes07] ilustra sobre el código de un "*saboteur*".

```
public void testDisplayCurrentTime_exception()
throws Exception {
// Inicialización de fixture
// Define e inicializa el Stub
TimeProvider testStub = new TimeProvider()
{ // Test Stub (saboteur)
public Calendar getTime() throws TimeProviderEx {
throw new TimeProviderEx("Sample");
}
};
// "Instancia" el SUT
TimeDisplay sut = new TimeDisplay();
sut.setTimeProvider(testStub);
// Ejercita al SUT
String result = sut.getCurrentTimeAsHtmlFragment();
// Verifica salida directa
String expectedTimeString =
"<span class=\"error\">Invalid Time</span>";
assertEquals("Exception", expectedTimeString, result);
}
```

Fig. 4- 13: Código y uso de un *Stub* "*saboteur*" [Mes07].

- ***Mocks:*** Son objetos que reemplazan a objetos "colaboradores" en los cuales se pueden establecer expectativas acerca de las llamadas que esperan recibir, a las que responderán según sean dichas llamadas. El código de la unidad bajo prueba (SUT) puede hacer llamadas a los métodos del *mock*, quien devolverá resultados que fueron inicializados por las pruebas [MH04].

  Implementan la misma interfase del objeto del cual el SUT depende. Se comportan durante la fase de ejercitación de la prueba de manera similar a *stubs* y durante tal período el SUT cree estar comunicándose con sus verdaderos "colaboradores". La principal diferencia radica especialmente en las fases de inicialización y verificación.. En la fase de inicialización se establece el tipo de llamadas que cada *mock* debe esperar recibir, así como la respuesta a brindar en cada caso. En la fase de verificación se procederá a comprobar que dichas expectativas hayan sido correctamente satisfechas; de forma de poder verificar que se hubieran invocado los métodos deseados y

que la respuesta fuera la indicada. La propuesta original acerca de *mocks* corresponde a Mackinnon, Freeman y Craig [MFC01].

Existen diversas bibliotecas para crear y manipular *mocks* (y *stubs*) en forma dinámica bajo xUnit, lo que viabiliza la utilización de estos símiles. Entre ellas se encuentran EasyMocks [Eas08] y Jmocks [Jmo08].

En la figura 4-14, según ejemplo resumido desde [Jmo08], se ilustra acerca de la utilización de Jmock en una prueba para un sistema de mensajes donde se desea probar al editor (*publisher*) en el envío de un mensaje a un suscriptor (*suscriber*). En dicho ejemplo el programa para ejecución (*runner*) de Jmock verificará que el símil del suscriptor fuera invocado como se esperaba. En caso que la llamada esperada no se hubiera producido, marcará la prueba como fallida.

```
// Interfase del suscriptor
interface Subscriber {
    void receive(String message);
}
// Prueba del Editor
Import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.jmock.Expectations;

@RunWith(JMock.class)
class PublisherTest {
    Mockery context = new JUnit4Mockery();

    @Test
    public void oneSubscriberReceivesAMessage() {
        // set up
        final Subscriber subscriber = context.mock(Subscriber.class);

        Publisher publisher = new Publisher();
        publisher.add(subscriber);

        final String message = "message";

        // expectations
        context.checking(new Expectations() {{
            one (subscriber).receive(message);
        }});

        // execute
        publisher.publish(message);
    }
}
```

Importa clases del marco Jmock para Junit 4

Crea el contexto “Mockery” donde el editor existirá.

Crea símil del suscriptor.

Crea el editor a probar, y registra el simil del suscriptor.

Crea mensaje a publicar

Método que se aguarda sea invocado.

Fig. 4- 14: Código y uso de Mocks [Jmo08].

En la figura 4-15 se expone un nuevo ejemplo, extraído de [Fow207], que muestra un fragmento de código correspondiente a la utilización de *mocks* para el mismo ejemplo precedente de la figura 4-13 sobre *stubs*. En este caso se reemplaza tanto al proveedor de servicio de correo como a la clase denominada *warehouse*. Se utiliza la biblioteca Jmock.

```
Class OrderInteractionTester...
public void testOrderSendsMailIfUnfilled() {
  Order order = new Order(TALISKER, 51);
  Mock warehouse = mock(Warehouse.class);
  Mock mailer = mock(MailService.class);
  order.setMailer((MailService) mailer.proxy());
  mailer.expects(once()).method("send");
  warehouse.expects(once()).method("hasInventory")
    .withAnyArguments()
    .will(returnValue(false));

  order.fill((Warehouse) warehouse.proxy());
 }
}
```

Fig. 4- 15: Código y uso de Mocks [Fow207].

4.3.3 Consideraciones sobre utilización y diseño

De acuerdo con Fowler [Fow207] la distinción esencial entres las técnicas de sustitución proviene de la diferente aproximación a las pruebas efectuadas por quienes utilizan *mocks*, estableciendo dos dicotomías principales: una que tiene como eje la verificación de resultados y otra que se focaliza en la estrategia utilizada para las pruebas, la cual divide en dos grandes corrientes a quienes practican TDD. Sobre esta última dicotomía, que sitúa como clásica vs. "*mockista*", destaca que involucra estrategias de diseño diferentes, bajo TDD, más que meras formas de llevar adelante las pruebas; en tal sentido Fowler alinea con esta última estrategia al aporte efectuado por Dan North [Nor06], que le expande bajo BDD (3.3.3.1) junto al proyecto JBehave [Jbh07], reemplazo de JUnit que apunta a permitir la especificación de comportamientos, en la misma línea de proyectos tales como Rspec [Ast05] [Rsp05].

- Dicotomía estado vs. comportamiento:
    - Prueba usando verificación del estado: Según el patrón general de utilización de los xUnit, se debe establecer el conjunto de objetos básicos para efectuar la prueba, lo que incluye al SUT y sus colaboradores. Una vez efectuada la prueba deberá procederse a la verificación del estado según todos los objetos involucrados (SUT y sus colaboradores).
    - Prueba usando verificación de comportamiento: Siguiendo el patrón general se establecen diferencias fundamentales con respecto a la

verificación del estado, ya que no solo se inicializan los datos sino que también se establecen expectativas acerca de la forma en que deberán comportarse las interacciones con los "colaboradores", que son sustituidos por *mocks*. Al momento de verificar los resultados producidos por la ejercitación del SUT, según los estímulos brindados por la prueba, además de la verificación del estado con respecto al SUT también se procede a verificar que los *mocks* fueran invocados cumpliendo las expectativas establecidas.

Meszaros [Mes07] describe ambos componentes de esta dicotomía clasificándoles como dos patrones diferentes dentro de un conjunto mayor de patrones de verificación de resultados.

- Dicotomía TDD clásico vs. TDD "*mockista*":

    - TDD clásico: Según ese estilo se usan objetos reales siempre que sea posible y solo se recurre a "dobles", generalmente *stubs*, para el caso que la manipulación o creación de los objetos reales sea dificultosa, problemática o con efectos no deseables durante la prueba.
    - TDD "*mockista*": Según este estilo siempre que es posible se utilizan *mocks* para sustituir a los "colaboradores".

Fowler cita entre las ventajas de la aproximación "*mockista*" la posibilidad de guiar el desarrollo también en base a las expectativas suministradas, aunque cuando se programa en capas e interviene el desarrollo de una interfase gráfica destaca que el enfoque clásico parece apropiado. Sostiene que la utilización de este último estilo podría no proveer una adecuada aislación de las unidades a probar durante las pruebas unitarias y conlleva a una mayor complejidad en las *fixtures,* que puede resolverse con reutilización de código y con clases –fábricas- para creación de *fixtures* utilizando el patrón conocido como *Object Mothers*[36] [SP01]. El estilo clásico, con xUnit, sería más una prueba de mini integración que una prueba unitaria.

Según Meszaros el enfoque "*mockista*" utiliza el estilo de diseño NDD y la técnica *Mock Objetcs* (3.3.3.2) de Freeman et al. [Fre04] que le soporta, siendo un refinamiento de TDD para guiar el diseño de las interfases en base a los servicios que un objeto requiere, descubriendo y diseñando tanto roles como responsabilidades desde la acción de escribir las pruebas. El estilo NDD combina los beneficios de TDD con la elegancia conceptual del desarrollo tradicional "*top-down*", permitiendo diseñar y probar capa a capa [Mes07].

[36] *Object Mother* es un patrón cuyo propósito es generar objetos semejantes (*stubs*) a los reales que existirán en producción.

Capítulo 5

FIT: "*Framewok for Integrated Tests*"

En este capítulo se examina la herramienta para mejora de la comunicación en el desarrollo de *software* y automatización de pruebas de aceptación denominada "*Framework for Integrated Tests"* (FIT), resumiéndose adicionalmente acerca de extensiones e implementaciones de la misma.

Este capítulo está organizado de la siguiente manera:

- En la sección 5.1 se brinda una introducción.
- En la sección 5.2 se resumen sus características básicas.
- En la sección 5.3 se expone acerca de la extensión para flujos de trabajo.
- En la sección 5.4 se brinda información acerca de su arquitectura. enumerándose implementaciones de FIT, extensiones e integración.
- En la sección 5.5 se resumen consideraciones sobre su uso.

5.1 Introducción

FIT [Cun207] es una herramienta creada por Ward Cunningham que permite automatizar las pruebas de aceptación, concebida con el objetivo principal de mejorar tanto la comunicación como la colaboración en el desarrollo de *software*. El autor destaca especialmente estos dos últimos aspectos en su definición de FIT:

"*Buen software requiere colaboración y comunicación. FIT es una herramienta para mejorar la colaboración en el desarrollo de software. Es una manera inestimable de colaborar en problemas complejos y obtener la visión correcta, en forma temprana. FIT permite a los clientes, verificadores (testers) y programadores aprender lo que sus programas deberían hacer y lo que hacen. Automáticamente compara las expectativas de los clientes con los resultados actuales.*" [Cun207].

Los antecedentes de FIT se remontan al año 1989, cuando Cunningham implementó en *SmallTalk* la lectura y ejecución de casos de prueba codificados en planillas de cálculo electrónicas [Cun307], para luego continuar evolucionando hasta que finalmente FIT se libera en el año 2002 bajo licencia GNU-GPL [GNU91].

A continuación (secciones 5.2 a 5.4) se ofrecerá un resumen construido a partir del contenido del libro "*Fit for Developing Software: Framework for Integrated Tests*" [MC05] y con aportes de otras fuentes. Se utilizarán ejemplos suministrados en dicho libro pero codificados en lenguaje C# [Csh06], correspondientes a la versión de FIT portada a .Net por V. Lagsma [Vla06].

5.2 Características básicas

FIT utiliza un mecanismo dirigido por comandos y datos. Este mecanismo implementa los siguientes pasos básicos:

- Lee tablas que contienen tanto los parámetros para la prueba (datos, comandos) como los resultados esperados (ejemplos).
- Interpreta cada tabla con una clase (conocida como "*fixture*").
- Compara el resultado de correr el programa bajo prueba con los resultados esperados, resaltando coincidencias, discrepancias y excepciones.

Si bien fue concebida para pruebas de aceptación, puede considerarse agnóstica con relación al tipo de prueba.

<u>Acerca de las tablas:</u>

Las tablas que contienen los casos de prueba deben estar en formato HTML [W3C107] y el resultado se expresa en el mismo formato. Según Mugridge y Cunningham: "*FIT fue diseñado originalmente para trabajar con archivos HTML*" [MC05]. La especificación original del programa motor de ejecución de pruebas,

conocido como *FileRunner,* indica que se le puede suministrar tanto un solo archivo como un camino, de forma que lea todos los archivos contenidos en una estructura de directorios. Todo contenido que no sean tablas HTML lo ignora. FIT define formatos para las tablas, aplicables a diferentes escenarios, estableciendo también diversas convenciones, entre otras para la expresión de los resultados de las pruebas.

Acerca de las *fixtures*:

Las *fixtures* ejecutan objetos del sistema a probar (SUT). Implementan las pruebas a partir de la información que obtienen desde las tablas y se programan en el mismo lenguaje que el SUT. Esto último explica la necesidad de portar FIT a diferentes lenguajes para ampliar su campo de aplicación. La versión original fue escrita en Java; habiendo sido portado a diferentes lenguajes y plataformas. La figura 5-1 ilustra acerca de lo anteriormente expuesto.

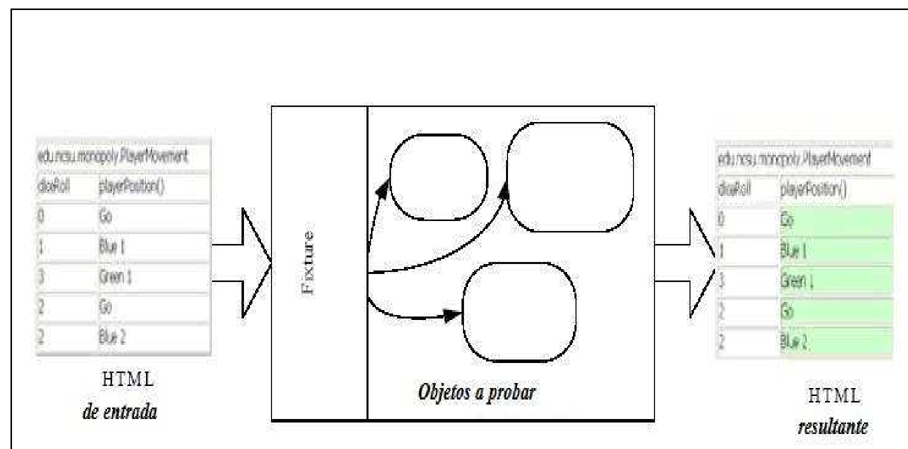


Fig. 5- 1: FIT: *fixtures* y tablas (NC State University).

Las *fixtures* puedan interpretarse como 'el pegamento' que une las tablas con los objetos a probar [MC05]. El código que los programadores deben escribir generalmente es breve, simple y utiliza clases que suministra FIT, también llamadas *fixtures.* Implementan el patrón *"Adapter"*.

Acerca de la metodología de trabajo

Los usuarios y verificadores expresan los casos de prueba mediante tablas con formatos preestablecidos, pudiendo combinarlas. Una vez que dichas tablas han sido pobladas con los datos necesarios, los desarrolladores escriben el código que las vincula con los objetos a probar. Es de destacar que de utilizarse EATDD (3.3.1) los desarrolladores escribirán también el código de la aplicación guiados por estos casos de prueba.

Para viabilizar la utilización de FIT es necesario establecer una estrecha colaboración entre los usuarios, verificadores y desarrolladores. Según Mugridge y Cunningham el formato tabular resulta natural a los diferentes actores, estableciéndose de esa manera

un mecanismo simple que facilita la comunicación y colaboración [MC05]. Este esquema de trabajo se representa en la figura 5-2.

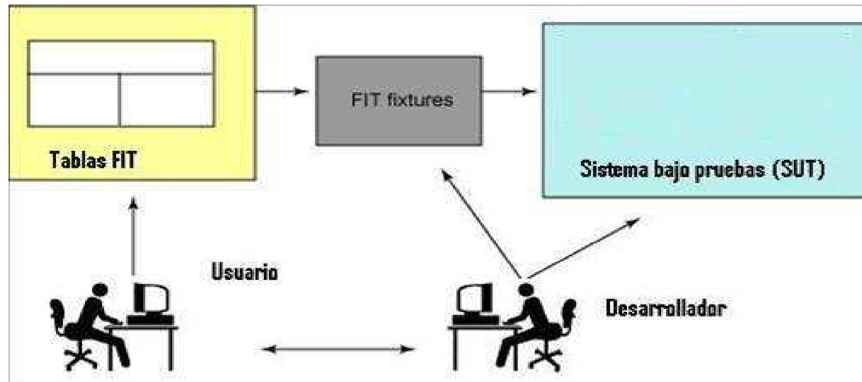


Fig. 5- 2: Esquema de trabajo con FIT (basado en figura de [Vet06]).

Acerca del resultado de las pruebas:

FIT expresa el resultado utilizando el mismo formato de tabla que contiene el caso de prueba correspondiente. Utiliza colores aplicados sobre los elementos de la tabla que contienen los resultados esperados y los resultados obtenidos, siendo su significado el siguiente: si es verde el resultado coincide con el esperado, si es rojo el resultado no coincide con el esperado, con amarillo se denota que ocurrió una excepción y con gris se indica que la prueba no se ejecutó. En los casos de no coincidencia muestra junto al valor esperado el valor obtenido. Si el valor esperado no fue especificado simplemente despliega el valor obtenido. Si ocurrieron excepciones brinda información pertinente sobre las mismas.

5.2.1 Tablas y *Fixtures*

FIT ofrece tres familias de *fixtures* básicas, interpretes de tablas de igual nombre. En la tabla 5-1 se relaciona dichas *fixtures* con su funcionalidad.

Nombre	Para probar:
ActionFixture	Secuencia de acciones
ColumnFixture	Cálculos
RowFixture	Resultados de consultas

Tabla 5- 1: *fixtures* básicas.

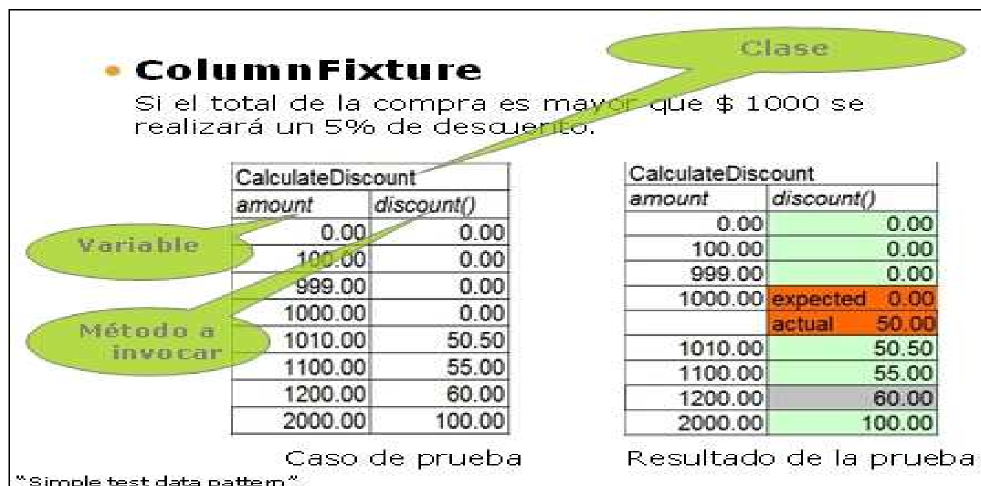
5.2.1.1 *ColumnFixture*

Sigue el patrón de pruebas "*Simple Test Data*" (2.5) [Cli04] y se utiliza para la verificación de cálculos. Define un formato de tabla específico del mismo nombre y una *fixture* (clase) que la interpreta.

La prueba consiste en ejecutar el programa bajo prueba una vez por cada fila de la tabla, suministrando para cada ejecución un juego de parámetros de entrada y

resultados esperados, obtenidos desde sus celdas. Al finalizar cada ejecución compara los resultados obtenidos con los esperados y expone el resultado de esa comparación, implementando un oráculo de entrada/salida (2.1.8).

En la figura 5-3 se ofrece un ejemplo, donde dado un requerimiento consistente en aplicar un descuento determinado si una compra excede cierto monto, la prueba consiste en verificar el cálculo de dicho descuento. A tales efectos se suministran, en cada fila de la tabla a partir de la tercera, diferentes valores del monto y del descuento esperado. La primera columna corresponde a una de valor dado ("suministrada") y la segunda es una de valor calculado ("calculada"). Los rótulos en la segunda fila identifican dichos valores, donde los símbolos () definen una columna "calculada".



CalculateDiscount	
amount	*discount()*
0.00	0.00
100.00	0.00
999.00	0.00
1000.00	0.00
1010.00	50.50
1100.00	55.00
1200.00	60.00
2000.00	100.00

CalculateDiscount	
amount	*discount()*
0.00	0.00
100.00	0.00
999.00	0.00
1000.00	expected 0.00
	actual 50.00
1010.00	50.50
1100.00	55.00
1200.00	60.00
2000.00	100.00

Fig. 5- 3: Caso de prueba utilizando *ColumnFixture* y resultado de la prueba.

Cuando es necesario expresar que se espera el rechazo a ciertos valores de prueba, sin provocar una excepción, se utiliza la palabra clave ***error*** en lugar del valor esperado en las columnas de resultados correspondientes. En este caso, de producirse la excepción al ejecutar, la celda aparecerá coloreada de verde. Para el caso del ejemplo equivaldría a la dupla: valor negativo en la columna *amount()* y palabra *error* en la columna *discount(),* para una misma fila.

Cuando es necesario expresar que no se desea especificar el valor de una celda se utiliza la palabra clave ***blank***; luego de la ejecución FIT devolverá el valor de retorno de la celda, si lo hubiera, sin efectuar comparación alguna. Si no se indica valor en una celda el marco entenderá que debe utilizar el valor de la celda para esa misma columna y la fila anterior.

Implementación

La primera fila de la tabla indica el nombre de la clase (*fixture*) que se ejecutará para implementar la prueba, la cual deberá ser programada extendiendo la clase del marco denominada *ColumnFixture*. Esta clase utilizada para implementar la prueba no

forma parte del código de producción del SUT. FIT "instanciará" dicha clase luego de leer la primera fila de la tabla[37].

Los valores en la segunda fila corresponden a nombres de variables de instancia públicas o a nombres de métodos públicos[38], según sean respectivamente columnas "suministradas" o "calculadas". Unos y otros deberán estar definidos en el código de la *fixture* suministrada por el desarrollador. Las variables públicas deberán contener los parámetros de entrada que se cargarán desde los valores contenidos en las celdas de las columnas "suministradas". Los métodos públicos invocarán los objetos a probar. FIT analiza los valores de la segunda fila para determinar el tipo de las columnas ("suministradas" o "calculadas") creando los objetos correspondientes (*typeAdapters*) que se encargarán de la adaptación del tipo de datos para cada columna.[39] Estos objetos serán usados para convertir los datos de las columnas "suministradas" al tipo correspondiente y asignarlos a las variables de instancia, o para ejecutar los métodos indicados por las columnas "calculadas", convertir el resultado esperado al tipo del dato de retorno y comparar contra el resultado obtenido. En caso de coincidencia se colorea de verde y en caso de discrepancia se colorea la celda de rojo, además de exponer el valor obtenido.

El marco utilizará notación *CamelCase*[40] para componer los nombres de las clases y métodos a partir de los nombres indicados en las tablas; ignorando tipografía y otros detalles decorativos. No se requiere que los nombres de los métodos y parámetros coincidan con los del SUT, por lo cual es posible utilizar el lenguaje del dominio del problema, permitiendo al cliente expresar la prueba sin necesidad de conocer acerca de la estructura de programación subyacente, siendo esto válido también para otras *fixtures*.

En el ejemplo de la figura 5-3 la *fixture* "*CalculateDiscount*" es una clase que contendrá el método público *discount*, donde se definirá una variable de instancia pública *amount* que sucesivamente se cargará con los valores correspondientes de cada fila. Ese método se ejecutará una vez por cada fila de la tabla. En la figura 5-4 se muestra el código en C# de dicha clase.

El código utilizado es sumamente simple:

- Define la clase cuyo nombre se muestra en la primera fila de la tabla, heredando de la clase *ColumnFixture*.
- Define las variables y métodos correspondientes a los rótulos de las columnas expresados por los valores en la segunda fila.
- Invoca el método que calcula el descuento en el SUT.

---

[37] En general, luego de leer la primer fila de una tabla, crea un objeto de la clase indicada con el nombre dado en su primer celda e invoca un método *doTable()* presente en el.

[38] Los métodos y variables se definen públicos, ya que justamente la intención es evitar el ocultamiento hacia el marco.

[39] Para cada columna "no calculada" su objeto *typeAdapter()* busca la variable de instancia correspondiente y su tipo. Si corresponde el rótulo a una columna "calculada" busca el método correspondiente y su tipo de retorno.

[40] Nombre dado en inglés a la práctica de escribir palabras compuestas, donde las palabras están unidas sin espacios y cada palabra comienza con una letra en mayúscula. http://es.wikipedia.org/wiki/CamelCase , http://c2.com/cgi/wiki?CamelCase.

La *fixture* básica *ColumnFixture* y otras clases del marco implementan el resto.

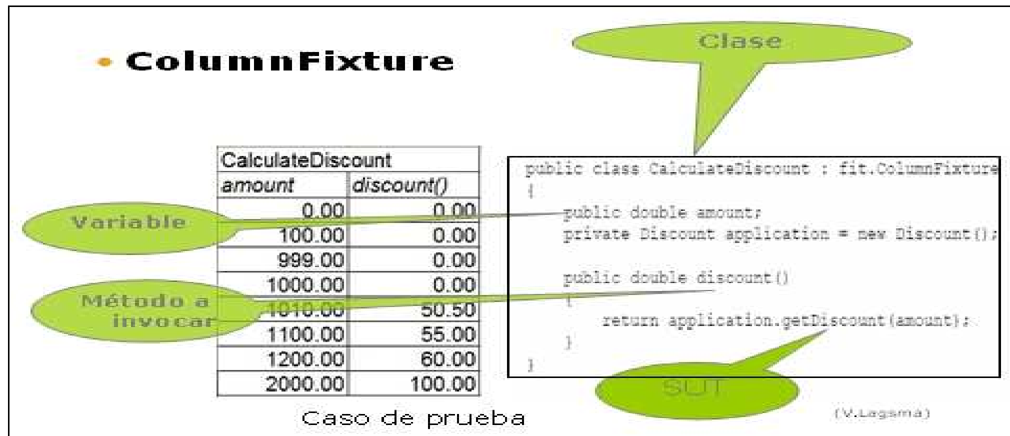


Fig. 5- 4 Tabla de tipo *ColumnFixture* y fragmento de código C# que la interpreta.

.

5.2.1.2 *ActionFixture*

Implementa patrones de pruebas del tipo "*Process*" (2.5) [Cli04] y se utiliza para verificar que un flujo de acciones sobre un sistema tenga el resultado esperado. Al igual que otras *fixtures*, define un formato de tabla específico del mismo nombre y una *fixture* (clase) que lo interpreta.

La prueba consiste en ejecutar acciones en secuencia y verificar los valores obtenidos, a partir de la definición de la prueba contenida en filas sucesivas de una tabla. Usa la metáfora de un dispositivo para controlar la prueba: se arranca el dispositivo, se ingresan valores en campos de entrada, se presionan botones y se comprueban resultados obtenidos contra resultados esperados. Para indicar estas acciones se vale respectivamente de los comandos *start*, *enter*, *press* y *check*. Se debe tener en cuenta que esta *fixture* ejecuta acciones que pueden tener efectos sobre el SUT.

El formato de las tablas de tipo *ActionFixture* prescribe que en la primera fila se indique su tipo. A partir de la segunda fila se debe incluir una sucesión en el siguiente orden: una fila donde se especifica *start* en la primera columna y el nombre del "actor" (clase) que comanda la prueba en la segunda, seguida de otras filas indicando en su primera columna las acciones *enter*, *press* o *check*. Es posible comenzar varias secuencias de este tipo (comenzando cada una con la palabra *start*) en una tabla.

En la figura 5-5 se ofrece un ejemplo con un flujo de trabajo y comprobación a realizar, mientras que en la figura 5-6 se expone parte del código C# suficiente para su instrumentación. Corresponde a los eventos que se expondrán a continuación.

- Enciende el dispositivo "acciones sobre el servidor de conversación" (*ChatServerActions*).
- Ingresa un nombre de usuario: *Anna.*
- Presiona el botón conectar para conectar a *Anna* al servidor de conversación (*chat server*).
- Ingresa el nombre *Lotr* para una sala de conversación (*chat room).*
- Presiona el botón de crear salas para crear la sala *Lotr.*
- Presiona el botón para ingresar el usuario, *Anna,* a la sala *Lotr.*
- Ingresa el nombre de usuario *Luke.*
- Presiona el botón conectar para conectar a *Luke* al mismo servidor.
- *Luke* se conecta al mismo servidor de conversación.
- Presiona el botón para ingresar un usuario, en este caso *Luke,* a la sala *Lotr.*
- Comprueba que en la sala *Lotr* hay dos usuarios.

Fig. 5- 5: Caso de prueba de tipo *ActionFixture* y resultado de la prueba.

El resultado de la ejecución se ofrece comparado el resultado obtenido versus el esperado, resaltando en color las condiciones de éxito o falla, en la fila que corresponde a la acción expresada por el comando *check.* De esperarse rechazo a una acción, sin provocar una excepción, se utiliza la palabra clave *error* en lugar del valor esperado. En este caso de verificarse la excepción la celda aparecerá coloreada de verde.

Finalmente es de destacar que *ActionFixture* no fue diseñado originalmente para comandar interfases gráficas sino para ejecutar métodos de una clase en una secuencia dada [MC05].

Implementación

La fila cuya primera columna contiene el comando *start* indica que el valor contenido en la segunda columna corresponde al nombre de la clase que FIT deberá

"instanciar". Dicha clase, denominada "actor", comandará la prueba hasta que aparezca otra fila con el comando *start*. No forma parte del código de producción del SUT, siendo una subclase de la clase básica *Fixture.* La clase actor, una vez "instanciada", arrancará la aplicación a probar creando el objeto necesario. De acuerdo al contenido de la tabla FIT disparará los métodos apropiados suministrados por esta clase. El resto de las filas entre dos *start* contienen en su primera columna un comando que puede ser *enter, press* o *check*. FIT buscará e invocará el método del actor que sea igual al valor en texto obtenido de cada segunda columna según se detallará a continuación.

- *enter:* Busca el método que tenga el nombre indicado y un parámetro. Determina el tipo del parámetro. Convierte el texto en la tercera columna al tipo del parámetro. Invoca el método con el parámetro.
- *press:* Busca e invoca el método que tenga el nombre indicado y sin parámetros.
- *check*: Busca el método que tenga el nombre indicado y sin parámetros, pero que retorne un resultado. Determina el tipo de datos de retorno. Toma el texto de la tercera columna, lo convierte a ese tipo y lo compara. Si coinciden marca la celda con verde. Si no coinciden la marca con rojo y expone el valor de retorno.

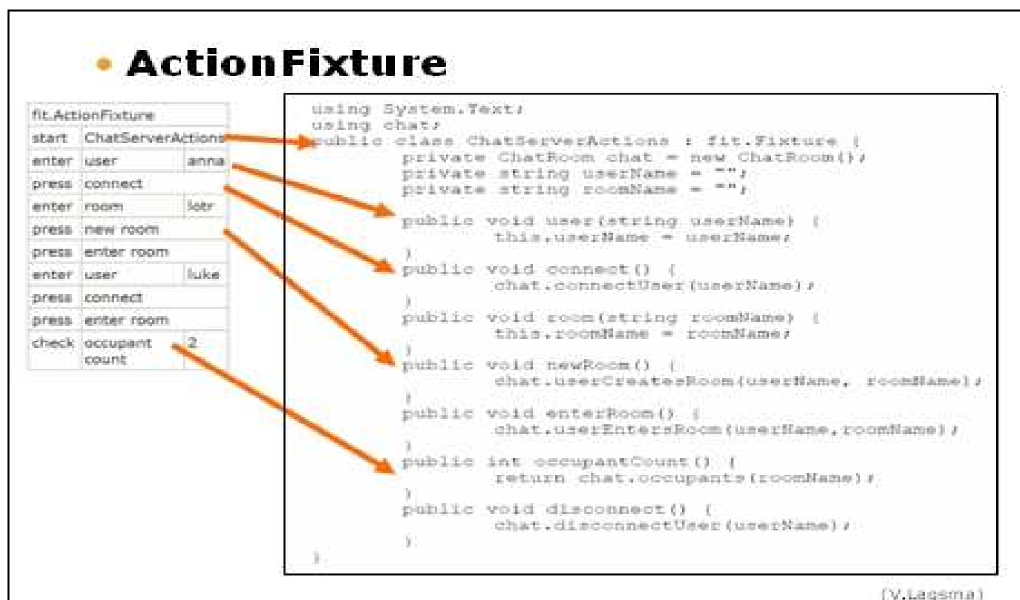


Fig. 5- 6: Caso de prueba de tipo *ActionFixture* y fragmento de código C# interprete.

5.2.1.3 *RowFixture*

Implementa patrones de prueba del tipo "*Collection Management*" (2.5) [Cli04] y se utiliza para verificar que el resultado de una búsqueda o consulta sobre el SUT tenga el resultado esperado. También se utiliza para comprobar que los elementos esperados en una secuencia o conjunto estén presentes en el SUT. Al igual que otras

fixtures, define un formato de tabla específico y una *fixture* (clase) que lo interpreta, del mismo nombre.

El formato de las tablas de tipo *RowFixture* prescribe que en la primera fila se indique su nombre[41], lo cual determinará la lista a comprobar del SUT. La segunda fila rotula las variables de la lista. A partir de la tercera fila se indican los resultados esperados. Estas filas en su conjunto forman un único grupo, a diferencia de las filas independientes de las tablas *ColumnFixture*.

En la figura 5-7 se ofrece un ejemplo de un caso de prueba, mostrándose a la izquierda la especificación y a la derecha uno de los resultados posibles de la ejecución. En este caso la prueba consiste en verificar que en la sala de conversación denominada *Lotr* los ocupantes sean los usuarios *Anna* y *Luke*.

Cada fila que contenga algunos elementos no coincidentes con un conjunto de resultados de la consulta en el sistema se mostrará en rojo, como es usual, indicando los valores obtenidos junto a los esperados; aquellas filas excedentes se destacan con la palabra *surplus* (exceso) junto al valor obtenido en la primera celda. Si no se encuentra una fila en el conjunto resultado se marca en rojo.

- **RowFixture**
  Los ocupantes de la sala de chat "Lotr" son los usuarios Anna y Luke.

OccupantList	
User	*Room*
anna	lotr
Luke	Lotr

Caso de prueba

OccupantList	
User	*Room*
anna	lotr *expected* [illegible] *actual*
Luke	Lotr
Peter *surplus*	Chatroom

Resultado de la prueba

Fig. 5- 7: Caso de prueba *RowFixture* y resultado de la prueba.

En el caso que no se requiera verificar si hay elementos excedentes puede utilizarse una *fixture* similar pero que no efectúa dicha comprobación, llamada *SubSetFixture*. Esta fixture es una de las contenidas en la biblioteca *FitLibrary* acerca de la cual se tratará en (5.3).

El orden de las tuplas no se tiene en cuenta. De ser necesario comprobar el orden se debe incluir una columna *Order* (ver figura 5-8).

---

[41] Es conveniente establecer alguna convención en los nombres que permita diferenciarlas fácilmente de las tablas *ColumnFixture*, cuyo formato es muy similar.

DiscountGroupOrderedList				
Order	future value	max owing	min purchase	discount percent
1	Low	0.00	0.00	0
2	Low	0.00	2000.00	3
3	Médium	500.00	600.00	3
4	Médium	0.00	500.00	5
5	High	2000.00	2000.00	10

Fig. 5- 8: Caso de prueba *RowFixture* indicando comprobar orden [MC05 fig. 5.7].

Implementación

La primera fila corresponde al nombre de la clase suministrada por el desarrollador para implementar la prueba. Dicha clase será instanciada por FIT. No forma parte del código de producción del SUT, siendo una subclase de la clase básica *RowFixture.* Para efectuar la prueba la clase deberá recolectar los datos entregados por el SUT y usarlos para su comparación con los suministrados en la tabla, creando una colección local. Este tipo de *fixtures* están previstas para manipular conjuntos de objetos.

Los métodos y variables se definen públicos. FIT ignorará la tipografía. En la figura 5-9 se muestra parte del código, en C#, relacionado al ejemplo de la figura 5-7.

```
public class OccupantList : fit.RowFixture
{
        private ChatServer chat = new ChatServer();

        public override Object[] Query()
        {
                ArrayList occupancies = new ArrayList();
                foreach (Room room in chat.getRooms())
                {
                        collectOccupants(occupancies, room);
                }
                return occupancies.ToArray();
        }
        public override Type GetTargetClass()
        {
                return typeof(Occupancy);
        }
        private void collectOccupants(ArrayList occupancies, Room room)
        {
                foreach (User user in room.users())
                {
                        Occupancy occupant = new Occupancy(room.getName(),
                                user.getName());
                        occupancies.Add( occupant);
                }
        }
}
```

Fig. 5- 9: Código de ejemplo para implementación de *RowFixture* [Vla06].

5.2.2 Secuencia de tablas

En muchos casos una simple tabla no es suficiente para expresar un caso de prueba, por lo que se necesitará una secuencia de tablas.

Implementación

Se resumirá un caso de implementación descrito en [MC05], que ejemplifica sobre la posible secuencia de pasos efectuada por FIT, dada una secuencia de tablas como la expuesta por la figura 5-10. Constituyen el caso de prueba un flujo de tareas determinado por una usuario Anna que se conecta al servidor de conversaciones, crea una nueva sala e ingresa a la misma; verificándose luego que sea la única ocupante; finalmente se desconecta Anna del servidor y se comprueba que dicha sala ha quedado vacía.

TestConnectAndDisconnect - Mozilla

fit.ActionFixture	
start	ChatServer2

- *Anna se conecta, crea una nueva sala de chat llamada Lotr, e ingresa a la misma.*

fit.ActionFixture		
enter	user	anna
press	connect	
enter	room	lotr
press	new room	
press	enter room	

- *Anna es la única ocupante de la sala Lotr.*

OccupantList2	
room	user
lotr	anna

- *Anna se desconecta.*

fit.ActionFixture	
press	disconnect

- *No hay ocupantes.*

OccupantList2	
room	user

fit.Summary	
counts	2 right, 0 wrong, 0 ignored, 0 exceptions
input file	C:\TestConnectAndDisconnect.html
input update	Mon Aug 09 15:23:32 NZST 2004
output file	C:\reports\TestConnectAndDisconnect.html
run date	Mon Aug 09 15:23:34 NZST 2004
run elapsed time	0:00.01

Fig. 5- 10: Secuencia de tablas (basado en [Mc05 fig.24-1]).

Cuando FIT lee la primera tabla crea un objeto de tipo *ActionFixture*. La primera acción invocada es *start,* por lo cual se crea un actor, en este caso "*ChatServer2*", el cual creará a su vez un objeto del SUT de clase "*ChatRoom*". Se guardará una referencia hacia el SUT en una variable estática de forma tal que las subsecuentes acciones mediante la *fixture ChatServer2*, según tablas 2, 3 y 4, se efectúen para el SUT dado.

La secuencia de acciones tomadas al ejecutar según las tablas se ejemplifica en la figura 5-11, donde los objetos af1, af2 y af3 refieren a los respectivos objetos de clase *ActionFixture* y ol1, ol2 a los objetos de tipo *RowFixture* según la clase *OccupantList2*.

En el ejemplo suministrado solo la primer *ActionFixture* tiene la acción *start*. Las demás trabajarán sobre el mismo SUT, ya inicializado. Se deben tener en cuenta al utilizar *ActionFixture* sus efectos colaterales sobre la aplicación a probar, por lo cual es importante ordenar las *fixtures* subyacentes en un flujo de trabajo de pruebas de forma que accedan a la misma aplicación bajo prueba [MC05].

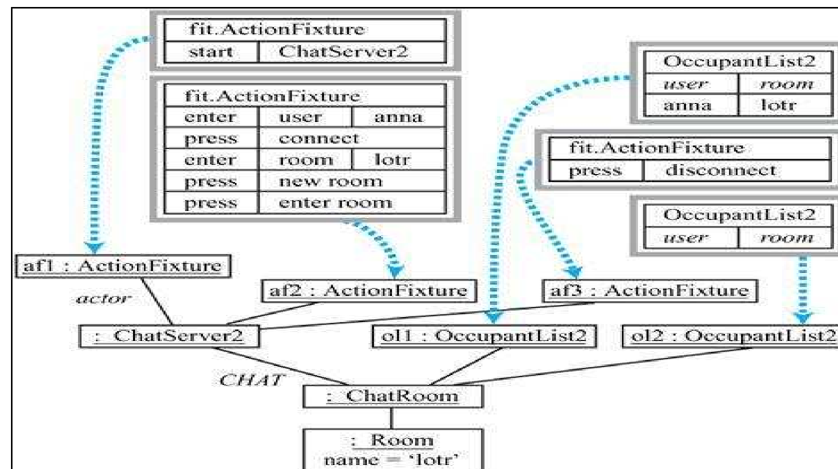


Fig. 5- 11: Fit ejecutando (extraída de [MC05 Fig.24.3]).

5.3 Flujos de trabajo

Los "flujos de trabajo" son un grupo especial de *fixtures* donde una sola *fixture* cubre las acciones en el resto de las tablas de la prueba y la secuencia de tablas se organiza diferente: solamente la primera contiene el nombre de una *fixture* [MC05].

Se agrupan en la biblioteca creada por Mugridge y conocida como *FitLibrary* [Mug107]. Entre las *fixtures* que contiene se destacan las siguientes:

- *DoFixture:* para probar acciones.
- *SetUpFixture:* para ingresar datos al comienzo de una prueba.
- *CalculateFixture:* para verificar cálculos.
- *ArrayFixture:* para verificar listas ordenadas.
- *SubSetFixture:* para verificar algunos elementos en listas desordenadas.

Se ofrecerá a continuación una síntesis acerca de *DoFixture* y *SetUpFixture.*

5.3.1 *DoFixture*

Esta *fixture* permite contemplar variados casos de pruebas. Brinda la posibilidad de expresar los casos de prueba utilizando una gramática construida a la medida del dominio, facilitando la construcción y adopción de un lenguaje ubicuo, el cual, según Mugridge y Cunningham, FIT ayudaría a crear naturalmente [MC05]. Cabe destacar que también se ha aprovechado esta *fixture* para lograr la integración con otras

herramientas. En la figura 5-12 se ofrece el mismo caso de prueba expuesto en (5.2.2) pero utilizando *DoFixture*.

Fig. 5- 12: Caso de prueba de tipo *DoFixture* y resultado de la prueba.

DoFixture recorrerá las filas de la tabla y ejecutará por cada fila un método. Construirá el nombre de los métodos a ejecutar concatenando los valores contenidos en las columnas impares, tomando como argumentos los valores contenidos de las columnas pares. En la figura 5-13 se muestra la interpretación que hace FIT del contenido de la fila de la tercer tabla como ***userCreatesRomm(anna, lotr)***. El resultado de la ejecución de los métodos se muestra con color, siguiendo la semántica ya definida anteriormente: si la acción es satisfactoria la marca en verde, si falla, aparecerá en rojo.

Fig. 5- 13: Interpretación de filas para formar nombres de métodos en *DoFixture*.

Se admite que en la primera columna de las filas se ingresen palabras reservadas que implican comandos, las cuales no serán tomadas en cuenta para la construcción del nombre del método. Se citarán las siguientes:

reject o ***not:*** Se espera el rechazo a una acción sin provocar una excepción.
check: Comprueba el valor de retorno del método versus el valor expresado en la última celda de la fila.
note: Ignora la fila.

Implementación

Esta *fixture* no se preocupa por las fronteras de las tablas. Puede ejecutar sobre cualquier número de tablas en sucesión, pero cuando encuentra la primera fila de una tabla hace una comprobación especial. Si dicha fila comienza con el nombre de una *fixture*, la utiliza en el estilo corriente de FIT, pero si contiene el nombre de un método que retorna una instancia de una *fixture*, entonces, de forma recursiva, usará dicha instancia para procesar la tabla, retornando a la *fixture* original *DoFixture* luego de finalizar la tabla [MC05].

En el caso del ejemplo, figuras 5-13 y 5-14, la clase *"ChatStart"* representa el nombre de una *fixture* de tipo "*DoFixture*". Este objeto será conocido como "objeto de flujo" (*flow fixture object*). El resto de los métodos en el ejemplo: *connectUser(), userCreatesRoom(), userEnterRoom()*, *userInRomm(), disconnectUser()*, pertenecen a dicho objeto de flujo. Todos ellos, salvo *userInRomm()*, disparan métodos en el SUT. El resultado de su ejecución se muestra en color sobre las celdas correspondientes a la acción. En cuanto al método *userInRomm(),* indicado en la primera fila de la quinta tabla, presenta la particularidad que retorna una *fixture*, en este caso del tipo *RowFixture*, con la cual FIT interpretará el resto de dicha tabla [MC05]. Finalmente es de destacar que el verbo "*Check"* se utiliza para verificar una acción.

```
public class ChatStart : fitlibrary.DoFixture
{
        private ChatRoom CHAT = new ChatRoom();
        public ChatStart()
        {
                base.mySystemUnderTest =  CHAT;
        }
        public bool connectUser(String userName)
        {
                return CHAT.connectUser(userName);
        }
        public bool userCreatesRoom(String userName, String roomName)
        {
                return CHAT.userCreatesRoom(userName, roomName);
        }
        public bool userEntersRoom(String userName, String roomName)
        {
                return CHAT.userEntersRoom(userName, roomName);
        }
        public Fixture usersInRoom(String roomName)
        {
         Collection<User> users = CHAT.usersInRoom(roomName);
                Object[] collection = new Object[users.Count];
                int i = 0;
                foreach (User user in users)
                {

                  collection[i++] = new UserCopy(user.getName());
                }
                return new ParamRowFixture( collection, typeof(UserCopy));
        }
        public bool disconnectUser(String userName)
        {
                return CHAT.disconnectUser(userName);
        }
        public int occupantCount( String roomName)
        {
                return CHAT.occupants(roomName);
        }
        public UserFixture connect(String userName)
        {
          if (CHAT.connectUser(userName))
                return new UserFixture(CHAT, CHAT.user(userName));
         throw new Exception("Duplicate user");
        }
         public fitlibrary.DoFixture room(String roomName)
        {
                return new fitlibrary.DoFixture(CHAT.room(roomName));
        }
        public bool roomIsEmpty(String roomName)
        {
                return CHAT.occupants(roomName) == 0;
        }
}
```
Fig. 5- 14: Ejemplo de implementación de *DoFixture* [Vla06].

La figura 5-15 ilustra acerca de la interacción entre FIT y las tablas al ejecutar según el ejemplo suministrado. El objeto de clase *ChatStart* es creado al leer la primera tabla, luego se disparan los métodos indicados por las tres tablas siguientes; mientras el método disparado por la acción de la quinta fila retorna una *fixture* que interpreta el resto de esa tabla.[42]

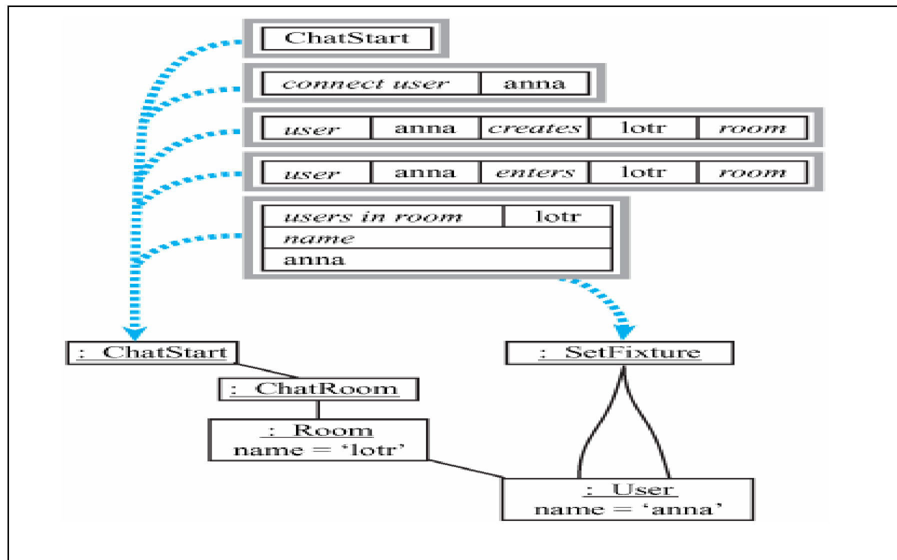


Fig. 5- 15: FIT ejecutando *DoFixture* [MC05 Fig.28-2].

5.3.2 *SetUpFixture*

Se utiliza para poblar colecciones o tablas de la base de datos al comienzo de una prueba. El uso normal es desde otras *fixtures* de flujo, como ser *DoFixture*.

Invoca el mismo método por cada fila de la tabla, pasándole como argumentos el contenido de sus celdas. Construye el nombre del método invocado concatenando los nombres de las columnas. Se encarga también de invocar dos métodos adicionales [43]: uno antes de procesar según la primera fila para efectuar la inicialización del caso de prueba (*setup*) y otro luego de ejecutar según la última fila para restaurar el ambiente (*tear down*).

La figura 5-16 muestra como luce una tabla para este tipo de *fixture* invocada dentro de un flujo. Luego de ejecutarse se colorea la tabla acorde al resultado de la sucesiva aplicación del método.

[42] De tipo *SetFixture* y suministrada por *FitLibrary*, similar a *RowFixture* pero más flexible.
[43] *Setup y TearDown.*

Discounts			
set ups			
future value	max balance	min purchase	discount %
low	0.00	0.00	0
medium	0.00	500.00	5
medium	500.00	500.00	3
high	500.00	2000.00	10
high	1000.00	500.00	5

calculate with		
owing	low purchase	future value discount
0.00	1000.00	0.00
1000.00	5000.00	0.00

ordered list			
future value	max owing	min purchase	discount %
low	0.00	0.00	0
medium	0.00	500.00	5
medium	500.00	500.00	3
high	500.00	2000.00	10
high	1000.00	500.00	5

subset			
future value	max owing	min purchase	discount %
low	0.00	0.00	0

Fig. 5- 16: Flujo incluyendo *SetUpFixture* [MC05 fig.28-6].

Implementación

El método indicado en la primera fila “instanciará” la *fixture*, especificando el SUT, o una subclase, pasándola al método invocador. Es una subclase abstracta de otra *fixture* conocida como *CalculateFixture*, que invoca a los métodos *SetUp()* y *TearDown()* para el proceso de inicialización y restauración respectivamente antes y después de procesar según las filas de la tabla. Es posible efectuar una sobrecarga de dichos métodos, por ejemplo para establecer una conexión a una base de datos al comienzo y desconectarse de la misma al final. La figura 5-17 muestra código C# para la implementación de la prueba soportada por las tablas referenciadas en la figura 5-16.

```
public class Discounts : fitlibrary.DoFixture
{
  private DiscountApplication app = new DiscountApplication();

 public Fixture setUps()
    {
      return new SetUpDiscounts(app);
        }
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------

public class SetUpDiscounts : fitlibrary.SetUpFixture
{
 private DiscountApplication app;

 public SetUpDiscounts(DiscountApplication app)
    {
    this.app = app;
       }
        public void futureValueMaxBalanceMinPurchaseDiscountPercent(
                 String futureValue, double maxBalance, double minPurchase,
                 double discountPercent)
     {
      app.addDiscountGroup(futureValue, maxBalance,
                           minPurchase, discountPercent);
      }
}
```

Fig. 5- 17: Código C# de ejemplo para *SetUpFixture* [Vla06].

5.4 Arquitectura e implementación

Esta sección resume sobre la arquitectura de FIT y algunas de sus implementaciones.

5.4.1 Arquitectura

Su arquitectura se basa en las siguientes clases [MC05]:

- ***Fixture*** se encarga del funcionamiento general de la ejecución sobre las tablas y provee de varios métodos. Las *fixtures* estándares extienden esta clase para procesar una tabla a su manera particular (ver figura 5-18).
- ***Parse*** se utiliza para representar las tablas y proveer retorno desde las pruebas.
- ***TypeAdapter*** se usa para las conversiones internas entre valores textuales en las celdas de las tablas y los tipos de datos requeridos por el lenguaje de implementación.

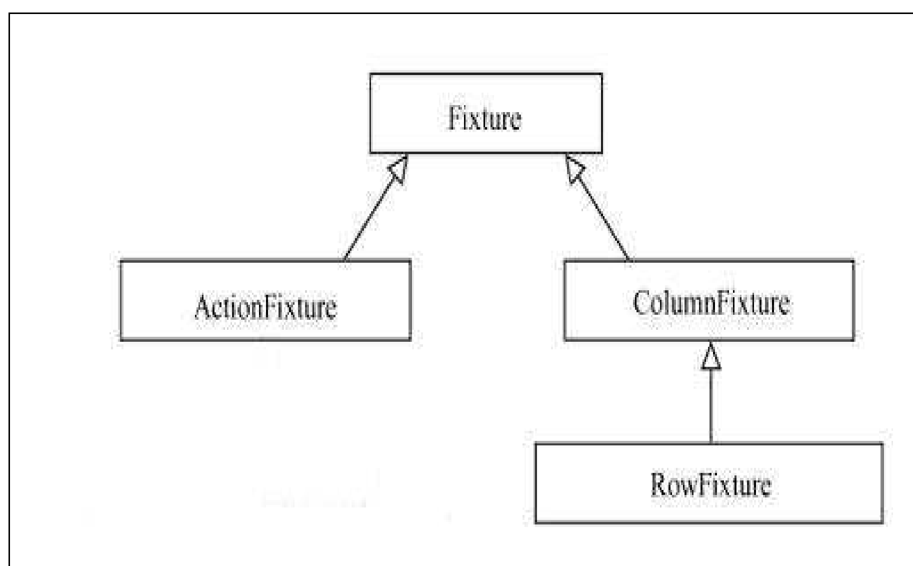


Fig. 5- 18: Diagrama de clases estándares de FIT (basado en [MC05 fig.37-5]).

5.4.2 Instalación y ejecución

La herramienta y la biblioteca *FitLibrary* pueden obtenerse descargando sus distribuciones desde Internet [FIT107] [Mug107].

FIT lee las pruebas desde archivos HTML. Para ejecutar sobre un único archivo HTML alcanza con escribir el siguiente comando: **Java –classpath fit.jar;. fit.FileRunner PruebaT.html Reporte.html** donde Prueba.html contiene las pruebas a ejecutar y Reporte.html contendrá el resultado de la ejecución. Si se desea ejecutar un conjunto de pruebas contenidas en un directorio, alcanza con alojar estos archivos en dicho directorio y correr FIT indicando el camino.

5.4.3 Implementaciones

Existen numerosas implementaciones para FIT que lo complementan brindándole, entre otras prestaciones, interfases para la ejecución y mantenimiento de las pruebas. Algunas están plenamente desarrolladas y otras en elaboración al momento de escritura de este trabajo. A modo de ejemplo se enumerarán a continuación algunas de dichas implementaciones o herramientas que utilizan FIT, mientras que su implementación más conocida y difundida, denominada FitNesse, se describirá en (C).

- WinfitrunnerLite [Sto05] Esta herramienta recolecta las tablas con las pruebas desde planillas electrónicas y permite ejecutarlas y visualizar los resultados bajo un IDE de aspecto similar a NUnit.
- Autat [SØ05] Herramienta específica para pruebas de aceptación de aplicaciones *web*. Surge para evaluar una alternativa con respecto a FitNesse y otras herramientas "*open source*" para la especificación de pruebas de aceptación de ese tipo de aplicaciones. Mediante un editor permite especificar y representar los casos de prueba en forma gráfica en lugar de tablas y se levanta como una extensión en Eclipse. Soporta otros marcos además de FIT.
- Storyteller [Mil206] Herramienta para crear y administrar pruebas automatizadas de código *.Net* basadas en el motor de FIT y FitNesse; creada para soportar ATDD y a futuro otros aspectos hoy débilmente abordados por FIT. Disponible solo en versión Alfa.
- ZiBreve [Mug207] Herramienta para crear y administrar las tablas con los casos de prueba, soportando la evolución de los relatos de las pruebas (*storytests*) y dando facilidades para la reconstrucción automatizada de las tablas interdependientes. Propone futura generación de código. Disponible bajo versión Beta. No se ha liberado código fuente.

Es posible integrar y extender FIT. En especial se citará a: "AntFit" [New04] para ejecutar como una tarea externa de Ant [Ant07], "Patang" [Jai106] para ejecutar en contenedores J2EE [J2EE], "Dbfit" [Adz207] como vehículo para implementar *"Test Driven Database Development"* y la propuesta [Ber03] de integración con Abbot[44] [Abb06] para utilizar TDD con elementos GUI de una aplicación.

5.5 Consideraciones

Entre las ventajas de FIT se destacan su soporte a la implementación de EATDD, la especificación de manera sustentable de los requerimientos, la utilización de un formato tabular que facilita la comunicación, una rápida curva de aprendizaje y el fomento de la colaboración entre desarrolladores, verificadores y usuarios. Ser una herramienta conducida por datos y comandos, de código libre y de simple instalación son también aspectos positivos destacables. Como debilidades se presentan el

[44] Abbot (*A Better Bot*): Herramienta que se integra a JUnit para probar componentes y programas JAVA con interfase gráfica.

costoso mantenimiento de los casos de prueba, la intensa reelaboración de *fixtures* ante los cambios en los casos de prueba y el SUT, la sobrecarga en los equipos de desarrollo y problemas de escalabilidad [Mel07] [Mil07] [MM07] [MRM04] [Rea05].

Es de destacar que FIT se ofrece como un "motor" de ejecución, estando fuera de su alcance la solución a otras carencias como el versionado, la interfase para definir las pruebas y la integración con el marco usado para el desarrollo. También es importante resaltar que se debe tener acceso a los fuentes del SUT[45] y conocimientos internos de la aplicación por parte del equipo de desarrollo.

En base a lo expuesto se entiende que para viabilizar la utilización de FIT sería necesario contar con marcos que aportasen un conjunto mínimo de facilidades, a saber:

- Editores de las tablas bajo control de tipos y formato.
- La visualización de los resultados, presente e histórica.
- Un repositorio de pruebas y resultados, con versionado.
- Una vinculación estrecha con el ambiente de desarrollo.
- La generación automática de código simplificando la tarea de creación y mantenimiento de *fixtures.*
- La reconstrucción automatizada de las tablas y la preservación de contenidos ante cambios en las mismas.
- La posibilidad de seleccionar que pruebas correr dentro de una suite.
- La posibilidad de detener pruebas que se encuentren en progreso.
- La integración con herramientas de seguimiento de fallas.
- La integración con herramientas de gestión de proyectos.

James Shore, coordinador del proyecto FIT, declara que es una herramienta bastante débil para pruebas [Sho07] especialmente basado en razones como las mencionadas precedentemente.

[45] Con FitNesse (C) es posible en ciertos casos levantar esta restricción.

Capítulo 6

GENEXUS

En este capítulo se describe acerca de GeneXus, la herramienta informática para la cual se plantea la propuesta contenida en esta tesis. La descripción ofrecida abarcará características básicas, enfocándose en particular sobre aquellas que serán referenciadas por la propuesta.

El capítulo está organizado de la siguiente manera:

- En la sección 6.1 se presenta la herramienta.
- En la sección 6.2 se describen características.
- En la sección 6.3 se enumeran herramientas complementarias.
- En la sección 6.4 se describe acerca de su nueva versión GeneXus X.
- En la sección 6.5 se resumen consideraciones sobre la herramienta.

6.1 Presentación

GeneXus [GX07] es una herramienta de especificación de sistemas de información que permite generar código para diversas plataformas y lenguajes [LSLN03]. Producida por la empresa Artech, su primera versión fue liberada al mercado para su comercialización en 1989. Sus creadores, Breogán Gonda y Nicolás Jodal, han sido galardonados con el Premio Nacional de Ingeniería en 1995, otorgado por el "Proyecto GeneXus" [GJ95].

Según consta en el documento "Visión general de GeneXus" [Art105]:

*"*GeneXus *es una herramienta inteligente…cuyo objetivo es asistir al analista y a los usuarios en todo el ciclo de vida de las aplicaciones. El diseño y prototipo es realizado en un ambiente windows…cuando el prototipo es totalmente aprobado por sus usuarios, la base de datos y los programas de aplicación son generados y/o mantenidos en forma totalmente automática, para el ambiente de producción".*

"GeneXus trabaja con conocimiento puro, lo que le permite realizar varias cosas: generar programas…, entender ese conocimiento… (no necesita documentación adicional …), y operar automáticamente con ese conocimiento..."

Según se expresa en "GeneXus: Filosofía" [GJ03]: *"GeneXus es, esencialmente, un sistema que permite una buena administración automática del conocimiento de los sistemas de negocios".*

6.2 Características

En esta sección se describirán características correspondientes a la novena versión de la herramienta.

6.2.1 Modelo de datos

Según establecen Gonda y Jodal en [GJ07] citando la clasificación de modelos de datos efectuada en el informe ANSI SPARC de 1978 [46]; la esencia y singularidad de GeneXus reposa en el "*modelo externo*", que contiene lo que realmente interesa a los usuarios y desarrolladores. Ese modelo externo es utilizado para obtener y almacenar el conocimiento, apoyándose en un súper conjunto[47] del modelo relacional para representar y manipular los datos. Todo el conocimiento contenido en el modelo externo es capturado automáticamente y sistematizado, asociándosele un mecanismo de inferencia y reglas generales independientes de una aplicación en particular.

[46] ANSI-SPARC: *Final report of the ANSI/X3/SPARC DBS-SG relational database task group (portal ACM, citation id=984555.1108830).* El informe referenciado introduce tres modelos: "Modelo Externo", que representa las visiones externas, "Modelo Conceptual", que contiene una abstracción de la realidad y "Modelo Físico" que representa el esquema de la base de datos.

[47] Se admiten grupos repetitivos y formulas.

Las características del modelo que están tratando de implementar son [GJ07]:

- Conocimiento adecuado para tratamiento automático.
- Consistencia.
- "Ortogonalidad".
- Proyecto, generación y mantenimiento automático de la base de datos y los programas.
- Escalabilidad.

Según se expresa en [Art105] no es posible construir un modelo estable de la realidad de la empresa, por lo cual se debe recurrir a un modelo incremental. Dado que ningún usuario posee una visión global e integradora de todos los aspectos en la empresa, se plantea construir dicho modelo incremental partiendo de las visiones que cada usuario tiene de los datos. Se explica en [GJ95] que básicamente estas visiones contienen todas las dependencias funcionales, de donde puede obtenerse el modelo normalizado por síntesis[48] en forma automática, que da lugar a la base de datos óptima y como subproducto las relaciones de integridad referencial necesarias para asegurar la consistencia. Según sus autores : "*puede demostrarse que, dado el conjunto de esas visiones de los usuarios, es posible construir una base de datos mínima que las satisface, la cual además es única; en ese estado el problema se transforma en un problema matemático y, entonces, es preciso resolverlo, para hallar esos datos*" [Art105].

Según Salvetto *"Los aportes fundacionales de Codd* [Cod70] *y J.D.Warnier constituyen ese modelo matemático. El problema a resolver consiste en la captura, integración y consolidación del conocimiento que contienen las visiones de los usuarios. formando un repositorio a partir de las visiones de los usuarios… se trata de más que un diccionario de datos; una base de conocimiento…"* Este problema es resuelto por GeneXus [Sal06].

Ken Orr puntualiza al respecto que la herramienta permite en forma automática lo que junto a J. D. Warnier concibieran a comienzos de los años 1970 y se formalizara como la metodología *"Data Structure System Development or Warnier/Orr"* [49], la cual postula una construcción de la base de datos a partir de la salidas que los usuarios desean obtener (visiones)[50]. Con GeneXus es posible ingresar estructuras de datos en pantallas de entrada con las cuales diseña en forma automática una base de datos normalizada generando el código para recorrerla, actualizarla y obtener reportes. También destaca que cuándo los requerimientos cambian o los usuarios presentan nuevos, permite establecer dichos requerimientos en el modelo y convertir automáticamente la base de datos previa al nuevo diseño y regenerar los programas

[48] Utilizando algoritmo de normalización por síntesis para llegar a 3ra Forma normal (3NF).
[49] Basada en los trabajos de J.D.Warnier publicados en "*Logical Construction of Programs*" Van Nostrand-Reinhold", 1974. y ,"*Logical Construction of Systems*", Van Nostrand-Reinhold, 1981, Ken Orr hace su propuesta metodológica conocida como método "Warnier/Orr", publicada originalmente en "*Structured Systems Development*" Yourdon Press, New York, 1977 y "*Structured Requirements Definition*" Ken Orr & Associates, Inc., Topeka, KS, 1981.
[50] Esta influencia ha sido corroborada por Gonda & Jodal en [GJ07].

que fueron afectados. Esto libera en gran medida a los desarrolladores de la preocupación sobre los futuros requerimientos y de los problemas de mantenimiento debidos al impacto que provocan los cambios y un modelo construido incrementalmente [HOC00].

6.2.2 Atributos

Las visiones de los usuarios se constituyen en la entrada básica y el marco de referencia lo constituyen los atributos. "*Cada atributo cuenta con un nombre, un conjunto de características y un significado*" [GJ07]. El significado de los atributos se corresponde al nombre dado a los mismos. El elemento clave entonces de esta tecnología es que un atributo tendrá el mismo nombre en todos los lugares donde aparezca y no hay dos atributos con significado diferente y con igual nombre. Esto se conoce como URA (*Universal Relationship Assumption*) [Art205]: "*Todo lo que es conceptualmente igual debe tener el mismo nombre y diferentes conceptos no pueden tener igual nombre*". A la adopción de la URA posteriormente se le sumó el concepto de subtipos de atributos y subtipos de grupos de atributos, dado que puede ser necesario dar diferentes nombres a atributos que representan el mismo concepto. Los atributos se convierten en la unidad semántica fundamental y el sustrato sobre el que se edifican los modelos GeneXus. El referir a atributos y no a tablas o archivos hace que sea posible proveer independencia de código con respecto a los cambios en la base de datos subyacente [GJ07].

Según lo expuesto el establecimiento de estándares de nomenclatura es sumamente importante como parte de la metodología de trabajo.

6.2.3 Variables

La Variables representan, al igual que en otros lenguajes de programación, áreas de memoria en donde se guardan valores en forma temporal durante la ejecución. Se distinguen de los atributos indicando su nombre precedido del símbolo **&**. Son locales a los objetos donde se definen y pueden estar basadas en atributos, de forma tal que hereden sus características.

6.2.4 Dominios

Los dominios permiten lograr uniformidad conceptual mediante la declaración de un tipo de datos y una dimensión que se aplique a un conjunto de atributos y variables que pertenezcan a un dominio. Un cambio futuro en la definición del dominio alcanza para que el mismo se propague a todo el conjunto de atributos y variables que lo integran [Marq06].

6.2.5 Base de conocimiento (KB) y modelos GeneXus

"*Una base de conocimiento GeneXus es un repositorio de conocimiento que contiene toda la información necesaria para generar una aplicación en múltiples plataformas. Consta de varios modelos:*

- **Diseño:** *Contiene los requerimientos de datos de la aplicación..*
- **Prototipo:** *Contiene la información de diseño específica para un ambiente de "prototipado". Uno o varios por base de conocimiento.*
- **Producción:** *Contiene la información de diseño específica para un ambiente de producción. Uno o varios por base de conocimiento"* [Art205].

En una base de conocimiento (KB) se captura el modelo externo y se lo sistematiza, en forma automatizada, poseyendo un mecanismo asociado de inferencia y de reglas de carácter general, como ser las correspondientes a la integridad referencial, independientes de una aplicación en particular [GJ07]. La realidad se describe mediante un conjunto de instancias de "Objetos Tipos" denominados "Objetos GeneXus" (6.2.10) contenidos en la KB.

6.2.6 Distribución y Consolidación

GeneXus ofrece la posibilidad de distribuir y consolidar (exportar e importar) objetos desde una KB a otra. La información a distribuir se compila en un archivo de texto con formato XML.

6.2.7 Metodología

GeneXus basa su propuesta, como ya se expresó, en modelar la realidad a partir de las visiones de los usuarios. La metodología asociada es una metodología ágil que promueve el desarrollo iterativo e incremental, apoyándose en un componente muy fuerte de automatización.

La metodología asociada a GeneXus consta de cuatro grandes etapas [Sal06]:

- **Diseño:** Se ingresan las visiones de los usuarios en un trabajo conjunto con ellos. Los objetos GeneXus utilizados para esta tarea tienen representación visual, lo que facilita la participación de los usuarios.
- **Prototipación:** Se genera la base de datos y los programas para el ambiente de prototipación.
- **Prueba del prototipo** por parte de los usuarios.
- **Implementación:** Se genera la base de datos y los programas para el ambiente de producción.

Estas etapas se iteran dando lugar a ciclos como los que se ilustran en la figura 6-1.

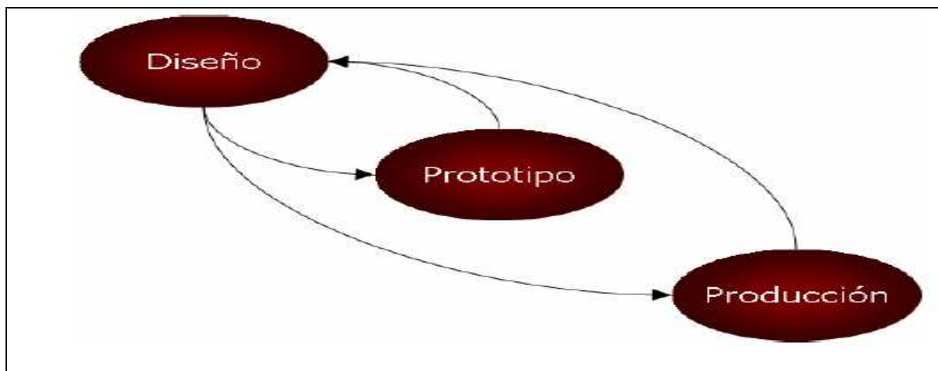


Fig. 6- 1: Metodologia: Ciclos Diseño-Prototipo y Diseño-Producción [Art105].

En las tareas de diseño el analista junto a los usuarios identifican y describen las visiones de datos. En todo momento se puede pasar a la fase de prototipo y obtener un prototipo completo, capaz de funcionar localmente, que recoge toda la funcionalidad incorporada al sistema, adhiriendo a un paradigma de especificación de requerimientos ejecutables[51]. Cuando se lo decide adecuado, se puede pasar a ciclos de diseño-producción donde se ejecuta la aplicación en un ambiente similar al de producción. Estos ciclos se iteran hasta alcanzar un modelo capaz de producir un producto de valor para el cliente.

La figura 6-2 ejemplifica la relación entre modelos y su ciclo[52].

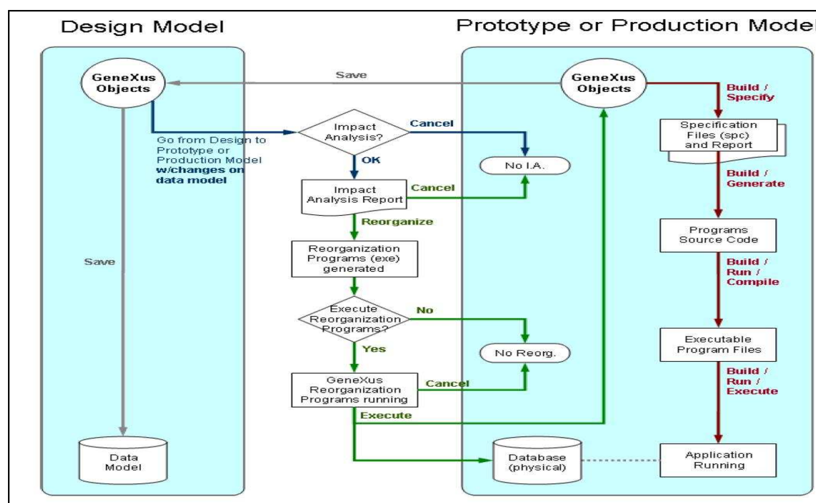


Fig. 6- 2: Modelos GeneXus [Art205].

Otros aspectos fundamentales son:

- **La definición de estándares de nomenclatura**, dada la importancia otorgada al nombre de los atributos (6.2.2).
- **La separación en módulos** ortogonales.

[51] Referido a especificaciones que pueden ser convertidas automáticamente en un prototipo [RBG98].
[52] Cuando se producen cambios GeneXus infiere un informe (análisis de impacto o *impact analysis report*) acerca del impacto que los cambios provocan; siendo capaz de propagarlos [GJ03].

- **La distribución del trabajo**[53] : Se posee una KB centralizada. Los desarrolladores del equipo trabajan sobre copias locales de la KB. Las modificaciones se distribuyen hacia la KB centralizada, donde se consolidan. Esto aplica también a equipos de trabajo dispersos.
- **Mantener KB separadas para desarrollo y producción:** Los cambios se realizan en una KB y una vez verificados y aprobados se impactan sobre la KB de producción.

6.2.8 Aplicaciones y Bases de Datos

GeneXus genera código en múltiples lenguajes para varias plataformas y bases de datos[54] a partir de los objetos contenidos en su KB. Es posible obtener diferentes combinaciones de plataformas, lenguajes y bases de datos para una misma KB.

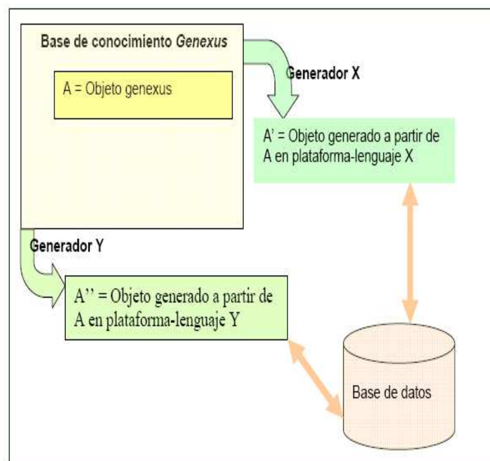


Fig. 6- 3: Generadores

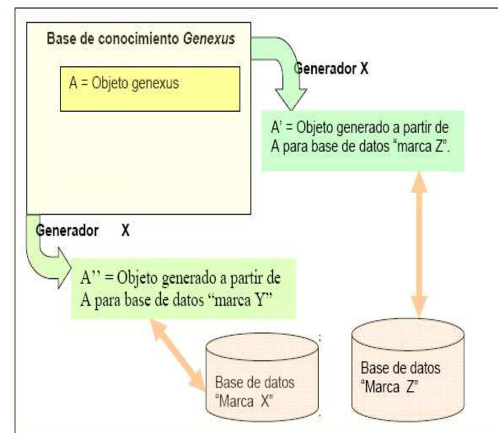


Fig. 6- 4: Múltiples bases de datos.

Para cada una de las combinaciones permitidas por la herramienta es posible construir un modelo dentro de la KB. GeneXus conoce la aplicación especificada en su KB en forma independiente a la tecnología, lo cual permite la independencia en relación a ella. Oculta muy bien las complejidades, no se necesita dominar cada tecnología soportada[55] así como tampoco se necesita saber acerca de los lenguajes o bases de datos de destino para construir una aplicación. Mediante el agregado de generadores se puede construir la misma aplicación para diferentes plataformas, lenguajes y bases de datos. Las nuevas versiones van incluyendo funcionalidades nuevas en respuesta a los avances tecnológicos que Artech considera importante incorporar.

Además de lo anteriormente expuesto, brinda al desarrollador un conjunto de sentencias y comandos de muy alto nivel, para declarar reglas y propiedades; un

---

[53] La versión 9 carece de administración de versiones. Por otra parte, una base de conocimiento accedida en modo de diseño permite solo a un desarrollador trabajar sobre ella. Estos aspectos son resueltos en la versión X (6.4).
[54] P/ej: Java, C#, VFP, VB, RPGII, COBOL. Plataforma Windows, Linux, j2ee, web.
[55] Por ejemplo J2EE, Ajax, Web, WebServices, Visual Foxrpo, Visual Basic, .Net., Java. Iseries, etc.

lenguaje declarativo y un lenguaje de cuarta generación "procedural". Permite, incluso en su lenguaje "procedural", que los programas fuentes se mantengan válidos aunque cambie la base de datos [GJ06].

6.2.9 Ambiente de desarrollo integrado

La interacción se efectúa mediante un IDE[56] conocido como "*GeneXus Development Environment*" desde donde se realiza la actividad con características de programación visual (VPL).

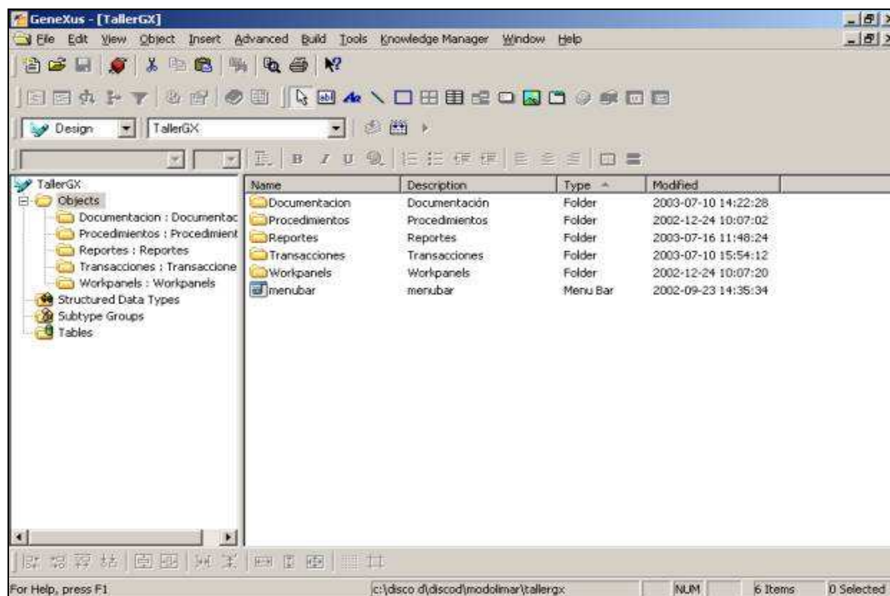


Fig. 6- 5: GeneXus Development Environment [Art205].

6.2.10 Objetos GeneXus

La realidad se describe mediante un conjunto de instancias de "Objetos Tipos" y las aplicaciones se construirán a partir de la creación de estas instancias en la KB. Estos objetos tipos constituyen un conjunto completo pero no cerrado [GJ07].

Entre los principales objetos se encuentran:

- *Transactions* (Transacciones).
- *WorkPanels – WebPanels* (Paneles de Trabajo *win* y *web*).
- *Procedures* (Procedimientos) y Reports (Reportes).

Algunos de estos objetos tipo tienen interfase gráfica representada por un formulario (*transactions, workpanels, webpanels*) y otros no cuentan con dicha interfase (*procedures*). Según sea se pueden definir eventos, reglas, una disposición (*layout*)

[56] *Integrated Develpment Environment* – Entorno integrado de desarrollo
http://es.wikipedia.org/wiki/Integrated_development_environment

para reportes y código. El desarrollador referencia atributos o variables dentro de los objetos.

Dado que las transacciones y los procedimientos son los dos elementos más importantes de GeneXus [GJ07] y considerando que serán directamente referenciados en la propuesta contenida en esta tesis, se brindará a continuación una introducción a los mismos. También se presentarán los "*Structured Data Types*"[57] (SDT) y los *"Business Components"*[58] (BC). Información detallada en estos tópicos se ofrece en [GDL07].

6.2.10.1 Transacciones (*Transactions*)

"*Entender las necesidades de los usuarios es una de las pocas tareas que no pueden automatizarse en el desarrollo de software, la metodología de GeneXus se basa en la descripción de las entidades de usuario final (objetos reales tangibles o intangibles) con las que la aplicación debe lidiar. Esto se logra describiendo las visiones que tienen dichos usuarios de estas entidades en un alto nivel de abstracción"* [Art105].

Las transacciones son los objetos que capturan las visiones de los usuarios, cuyo conjunto hace posible la creación de la base de datos única y mínima correspondiente. Representan entidades reales. A partir de esos objetos infiere el modelo de datos de la aplicación en 3NF y crea los objetos para altas, bajas y modificaciones de registros en la base de datos física.

Al definir las transacciones el desarrollador describe en forma explícita la interfase de usuario para la presentación y captura de datos, mientras que en forma implícita diseña el modelo de datos [Art205].

En las figuras 6-6 y 6-7 se muestran las pantallas para crear objetos de un cierto tipo y la utilizada para definir una transacción, respectivamente.

Fig. 6- 6: Creación de objeto [Art205].

Structure	Type	Description	Nulls	Formula
Invoice				
InvoiceID	Numeric(4.0)	Invoice ID	No	
InvoiceDate	Date	Invoice Date	No	
CustomerID	Numeric(4.0)	Customer ID	No	
CustomerName	Character(20)	Customer Name	No	
Level1	Level1Item	Level1		
ProductID	Numeric(4.0)	Product ID	No	
ProductName	Character(20)	Product Name	No	
ProductPrice	Numeric(8.2)	Product Price	No	
LineQuantity	Numeric(4.0)	Line Quantity	No	
LineTotal	Numeric(8.2)	Line Total		ProductPrice * LineTotal
InvoiceSubtotal	Numeric(8.2)	Invoice Subtotal		SUM(LineTotal)
InvoiceTax	Numeric(8.2)	Invoice Tax		InvoiceSubtotal * .085
InvoiceTotal	Numeric(8.2)	Invoice Total		InvoiceSubtotal + InvoiceTax

Structure | Form | Web Form | Rules | Events | Help | Documentation

Fig. 6- 7: Definición de una transacción [Art205].

[57] Tipos de datos estructurados.
[58] Componentes de negocio.

Como se mostró en la figura 6-7, la estructura de la transacción se define declarando todos sus atributos, los niveles de la misma y su clave. Algunos de los atributos pueden ser definidos como formulas, es decir que su valor se calculará a partir del valor de otros atributos y se comportarán como un atributo 'normal' a otros efectos. Dichas fórmulas son globales, por lo cual quedan ligadas al atributo. En la figura 6-8 se muestra la pantalla para definición de atributos.

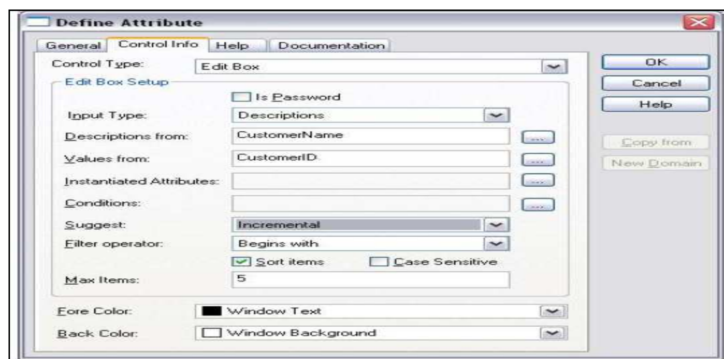


Fig. 6- 8: Definición de un atributo [Art205].

La pantalla donde se definen las transacciones (figura 6-7) presenta pestañas correspondientes al formulario *windows*, el formulario *web,* el editor para las reglas que serán expresadas en un lenguaje genérico y declarativo para hacer cumplir los controles sobre los datos a ingresar o eliminar, el editor para codificar los eventos asociados a elementos del formulario o al inicio o salida de la transacción, la ayuda y la documentación. Para cada transacción se crean en forma automática los formularios por defecto a los que se aplican temas y estilos. Se brindan ejemplos en las figuras 6-9 y 6-10.

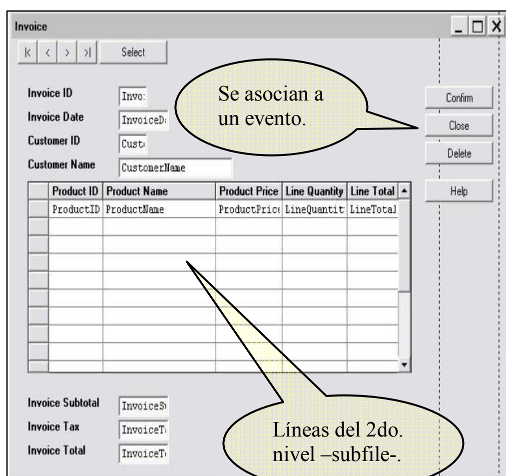


Fig. 6- 9: Formulario *windows* de la transacción [Art205].

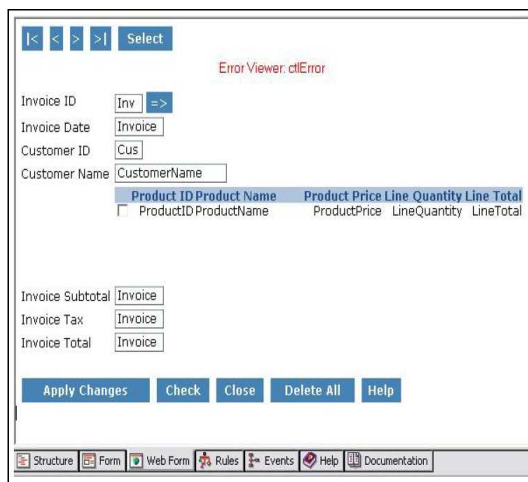


Fig. 6- 10: Formulario *web* (por defecto) de una transacción [Art205].

Eventos[59]

Contienen código que se asocia a elementos de la pantalla, a un menú de acciones, o a un momento del proceso.

```
// El código asociado a cada evento se señala con un marcador de
// comienzo y  otro de fin. El de comienzo es Event seguido del nombre del
// evento, el de fin es End Event. En este caso se muestran los marcadores
// predeterminados para ejecutar código por única vez al comienzo de la transacción.

        Event Start
                …………..
        EndEvent
```

Reglas

La lógica del negocio se expresa mediante reglas. Se escriben en forma declarativa. GeneXus infiere el orden y momento de su ejecución. Se brinda a continuación brevísimo ejemplo.

```
// Sintaxis: Rule [if condition] [on event…] [level att…];
Default(Invoice_date, today()); // carga el atributo Invoice_date con la fecha del día
// Despliega un error e impide ingresar la línea correspondiente si se
// ingresa un valor en el atributo ProductPrice menor a $5.
Error('No se permite ingresar precio inferior a $ 5") If ProductPrice < 5;
```

Estilos y Temas

Es posible definir estilos aplicables a objetos y formularios, eso permite que características comunes de diseño y código se definan en un único punto y se apliquen a los objetos que se declaren como basados en los mismos. A su vez también pueden definirse temas para aplicar a los formularios *web* en forma automática, a los efectos de lograr una presentación visual homogénea y de buen diseño. No se limita solo a transacciones, rige también para otros objetos.

Programa generado: Integridad referencial, valores límites, *subfile*

GeneXus generará a partir del conocimiento suministrado por las definiciones de las transacciones, las inferencias que pueda realizar y las reglas generales a aplicar, los programas necesarios en el lenguaje de destino para la creación y mantenimiento automático de las tablas de la base de datos. En particular, a partir de cada objeto transacción, generará los programas para que el usuario realice el mantenimiento de la información en forma interactiva sobre las tablas, incluyéndose el control de la integridad referencial y los valores límites declarados para los atributos en forma automática, sin necesidad de programación adicional. También se incluirá, entre

[59] Los eventos, reglas, estilos y temas no son privativos de las transacciones, otros objetos las incorporan también.

otras funcionalidades, las referidas a la carga, orden y paginación en el *subfile* (ver figura 6-9).

Un cambio en un atributo se propagará a todas las variables basadas en el mismo (al momento de su creación). Los objetos que utilicen el atributo o sus derivados se actualizarán en consecuencia mediante una reconstrucción automatizada.

6.2.10.2 Transacciones: Propiedad "*Business Components*"

Los objetos de tipo transacción pueden definirse de forma tal que las reglas de negocio, fórmulas y la integridad referencial queden 'incrustadas' en los mismos, con alto acoplamiento con la interfase gráfica. Esto utilización no es aconsejable ya que, entre otras razones:

- Atenta contra la reutilización.
- Obliga a probar las reglas vía la interfase gráfica.
- Impide centralizar su definición en un único punto en el modelo.
- Se transforma en fuente de errores al forzar su escritura redundante: si otro objeto actualiza datos el desarrollador deberá nuevamente escribir las reglas o su análogo "procedural"; debiendo además programar, en caso que los objetos que actualicen los datos sean procedimientos, los controles de integridad referencial.

Los *Business Components* (BC) se utilizan a los efectos de globalizar y centralizar reglas del negocio, integridad transaccional y formulas definidas en una transacción. Permiten asociar al representante del objeto del universo que se modela las reglas, las restricciones de integridad necesarias y las fórmulas, que se validarán y calcularán cada vez que se ejecute una acción sobre el.

Sus características más importantes son:

- Permiten utilizar la integridad referencial, las fórmulas y las reglas del negocio de una transacción sin utilizar su formulario; de esta forma pueden ser invocados por otros objetos para actualizar datos o a través de *Web Services* [W3C04] o *Enterprise Java Beans* [JCP07].
- Exponen propiedades y métodos.
- Se definen mediante una propiedad en las transacciones. Toda transacción tiene una propiedad "*Business Component*" que permite que sea invocada sin recurrir a su interfase gráfica (modo "silencioso") [GCW307].
- Todas las reglas que no están vinculadas a la interfase de usuario se disparan.
- Los eventos se ignoran, salvo el de comienzo y fin de la transacción. En dichos eventos se ignoran las referencias a otros objetos con interfase gráfica.
- Los errores se manejan mediante listas de mensajes construidas sobre tipos de datos estructurados (SDT), descritos más adelante (6.2.10.3).

En la figura 6-11 se muestran propiedades de una transacción y su selección como BC. En la figura 6-12 se muestra como se asigna una variable a un BC. Se dice que esa variable es de tipo BC y será la referencial del BC en otros objetos. La figura 6-13 brinda ejemplos de codificación de acciones sobre un BC.

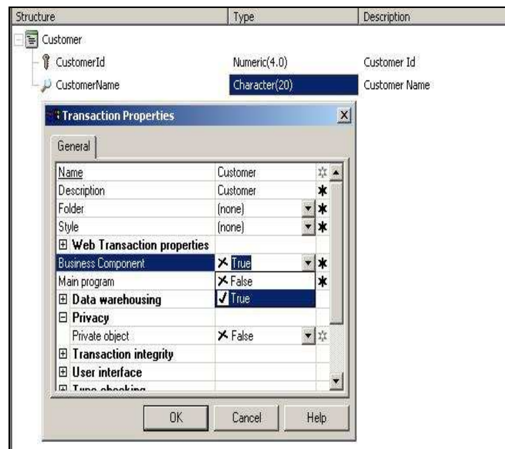


Fig. 6- 11: Business Component [GCW307].

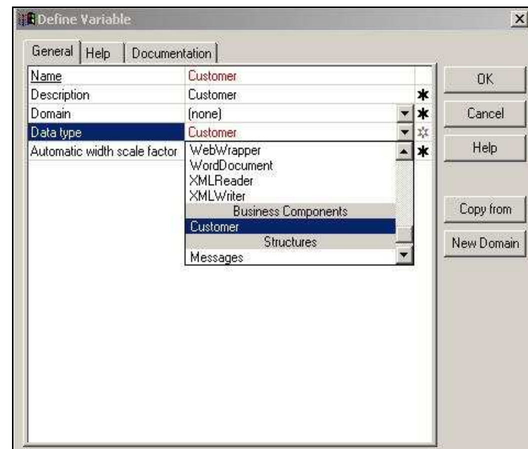


Fig. 6- 12: Variable de tipo BC [GCW307].

```
•   Alta de cliente

      &Customer = new Customer()  // Nuevo cliente
      &Customer.CustomerId = 123  // Clave
      &Customer.CustomerName = "John Smith" // Nombre
      &Customer.Save() // Inserta cliente

•   Alta o modificación de cliente

    &Customer.Load(123) // Carga el cliente de clave 123
    If &Customer.Fail()    // Si no existe el cliente de clave 123
            &Customer = new() // Nuevo cliente
     &Customer.CustomerId = 123 // Clave
    Endif
    &Customer.CustomerName = "Jorge Fosatti" // Nombre
    &Customer.Save() // Inserta o Modifica Cliente
```

Fig. 6- 13: Codificación de acciones con un BC [GCW307].

6.2.10.3 SDT: Tipo de Datos Estructurado

El objeto SDT permite definir estructuras de datos complejas formadas por varios elementos [GDL07]. Entre los usos posibles se destacan:

- Facilitar el pasaje de parámetros (en particular para uso en *Web Services*).
- Simplificar la lectura y escritura automática de XML (*eXtensible Markup Language*) [XML06].
- Mejorar la legibilidad del código.
- Permitir el manejo de listas de largo variable de elementos.

El objeto SDT tiene asociado un conjunto de propiedades y métodos. Se podrán definir variables, en cualquier objeto, basados en un objeto estructurado (SDT) o una estructura a su vez definida en uno. Márquez [Marq06] establece una analogía, a los efectos didácticos, entre un SDT y una clase en programación orientada a objetos,

donde el SDT se puede interpretar como la clase y su utilización en ejecución como una instancia de dicha clase.

Definición y Estructura

Se definen a partir de la pantalla mostrada en la figura 6-6. En la figura 6-14 se muestra la pantalla del editor de la estructura SDT. Esta estructura puede ser de múltiples niveles y es posible derivarla desde una transacción.

Name	Datatype	Collection
Cliente		
Nombre	Character(20)	
Identidad	Documento	
Nacionalidad	Character(20)	
Direccion		
Empresa	Character(20)	
Personal	Character(20)	
Telefonos	Numeric(10.0)	True

Fig. 6- 14: Estructura de un SDT [GDL07].

Un SDT puede constar tanto de elementos simples como compuestos, o ambos.

- Elementos simples: Se identifican unívocamente por su nombre y poseen un tipo de datos que puede ser un tipo básico, dominio u otro SDT. Pueden poseer una propiedad *Collection* si el elemento tiene múltiples instancias. En la figura 6-14 son simples: Identidad, Nacionalidad, Empresa, Personal, Teléfonos (colección).
- Elementos compuestos: Definen un nuevo agrupamiento de elementos simples. En la figura 6-14 son elementos compuestos los identificados como “Cliente” y “Dirección”.

Operador New

Retorna una instancia inicializada del SDT.
Ejemplo de sintaxis: &A = *New Client()*

Métodos

ToXML: Retorna una sarta con formato XML de los datos de la variable SDT.
Ejemplo de sintaxis: *&a = &b.ToXml()*

```
<Cliente xmlns = “name_Kb”>
  <Nombre>Acme</Nombre>
  <Identidad>11111111</Identidad>
  <Nacionalidad>Uruguay</Nacionalidad>
  <Direccion>
     <Empresa>Calle 18</Empresa >
     <Personal>Rambla 20</Personal>
  </Direccion>
  <Telefonos>
     <item> 1234567 </item>
     <item> 2224568 </item>
 </Telefonos>
```

Fig. 6- 15: XML a partir de SDT.

FromXML: Recibe una sarta como parámetro conteniendo una estructura XML desde la cual se carga la variable basada en el SDT. Ejemplo: *&b.FromXml(&a)*

Clone: Crea una nueva área en memoria y copia los datos de una variable en esta.

Colecciones

Las colecciones son elementos con varias instancias. Existe un conjunto de métodos para aplicar a colecciones, como ser *Add(Item, Position), Remove(Position), Clear().*

6.2.10.4 Procedimientos (*Procedures*) y Reportes (*Reports*)

GeneXus en sus orígenes era puramente declarativo. Esto no permitía generar el 100% de la aplicación al no estar resuelto el problema de la independencia del código “procedural” con respecto a los cambios en la base de datos. Finalmente ese problema fue resuelto [GJ06]. Los procedimientos (*Procedures*) y los reportes[60] (*Reports*) son objetos que permiten realizar tareas sin la intervención del usuario. Se escriben en un lenguaje “procedural” simple, incluyendo sentencias de control y acceso a datos, refiriendo a atributos y variables. GeneXus infiere las tablas que utilizará y las fórmulas. Según consta en [Art205] sus características son:

- **Reportes**: Interrogan la base de datos para obtener un informe mediante un proceso no interactivo incapaz de actualizar la base de datos.
- **Procedimientos**: Son capaces de interrogar la base de datos para obtener un informe o actualizarla. Se utilizan para definir funciones y subrutinas.

Las figuras 6-16 a 6-19, tomadas de [Art205], muestran la secuencia de pasos para crear un reporte a partir de la estructura de una transacción.

[60] Los reportes estaban incluidos en las primeras versiones de GeneXus, sin contenido procedural.

Fig. 6- 16: Creación de objeto reporte.

Fig. 6- 17: Reporte a partir de una transacción.

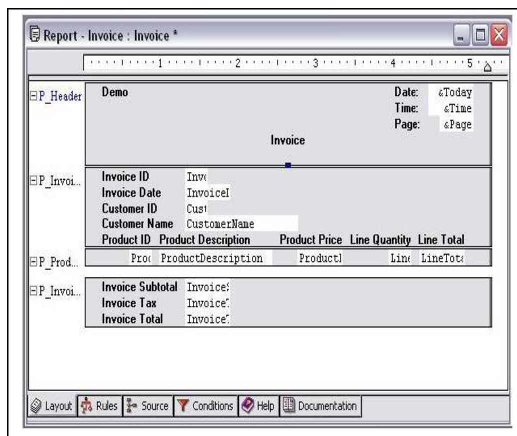


Fig. 6- 18: Disposición preliminar del informe.

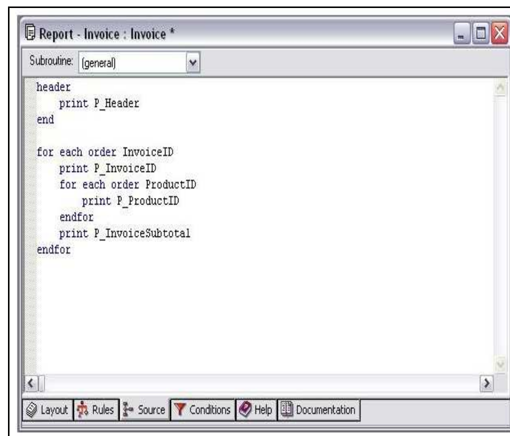


Fig. 6- 19: Código generado automáticamente.

El comando *FOR EACH*

En la figura 6-19 puede apreciarse el comando ***For Each***, en este caso anidado, que constituye el corazón del lenguaje "procedural" en GeneXus. Este comando recupera y actualiza información desde la base de datos. Se le indica la información a acceder mediante los atributos, y de esta forma se logra independencia del código con relación a la estructura de datos subyacente; siendo GeneXus quien infiere automáticamente cuales son las tablas apropiadas. Por cada *For Each* existe una tabla de modelo de datos asociada que puede ser "navegada" [61]. Esta tabla está compuesta por la tabla base y lo que se denomina su tabla extendida.

"*Dada una tabla base, su tabla extendida es el grupo de atributos que son directa o indirectamente determinados por la clave de la tabla base:*

- *Atributos que pertenecen a la tabla base*
- *Atributos que pertenecen a todas las tablas que están directa o indirectamente en una relación N a 1 con la tabla base"* [Art205].

[61] Recorrida. Incluye un plan de acceso con los filtros correspondientes y atributos a devolver.

El comando *CALL*

Con este comando se invoca un objeto desde otro, permitiendo pasarle parámetros (de entrada, salida o entrada-salida). Estos objetos invocados pueden ser programas externos. Cambiando propiedades de la invocación puede lograrse que la misma pase de ser una invocación a una subrutina del propio programa para convertirse en la invocación de un *Web Service*. Esta es una de las formas en que se oculta la complejidad y se permite el cambio de plataforma. La figura 6-20 brinda un ejemplo de uso. Existen también otras formas de invocación, como ser la función UDP, que admite n parámetros de entrada y devuelve un parámetro como salida; o el comando *CALL* utilizado desde una regla. El objeto invocado implementa la regla *parm* para indicar sus parámetros..

```
// Procedimiento llamador
// Invoca a un procedimiento llamado Customer
// pasándole como parámetro de entrada el nro. de clave
// -en este caso el número de cliente-. El procedimiento
// devuelve el nombre del cliente
// La P delante de customer indica el tipo de objeto invocado
// P-Procedimiento, T-Transacción, R-Reporte, etc.
// &Nro es una variable que contiene el número de cliente
// &Nombre contendrá el nombre del cliente
// &Nro y &Nombre son de igual tipo y dimensión que
// CustomerId y CustomerName
Pcustomer.call(in:&Nro, out:&Nombre)

// Procedimiento llamado
// En solapa rules:
Parm(in:&Number, out:&Name);
// En solapa layout:
// Pone nombre en blanco
&Name=nullvalue(&Name);
// Busca el nombre para el cliente solicitado
For Each
     Where CustomerId = &Number
      &Name = CustomerName
EndFor
```

Fig. 6- 20: Ejemplo de uso de comando Call.

6.2.11 Integridad Transaccional

Por defecto cada objeto GeneXus que actualiza la base de datos constituye una UTL (Unidad de Trabajo Lógica) [Marq06]; siendo posible también levantar esta restricción y definir las UTL con alcance de varios objetos. Para esto se indica el alcance de una UTL mediante la inclusión de instrucciones *commit* o *rollback*, o utilizando apropiadamente la propiedad "*commit on exit*"[62] de cada objeto.

6.2.12 Patrones

Gamma et al. plantean en [Gam94] la utilización del concepto de patrones aplicado al diseño de *software* orientado a objetos. Estos patrones de diseño permiten comunicarse y diseñar a un alto nivel de abstracción, estableciendo un vocabulario común de nombres, así como sistematizar la solución de problemas en ciertos

[62] Esta propiedad indica si debe o no procederse a efectuar la persistencia al salir de la ejecución de un objeto.

contextos, brindando una plantilla que puede ser aplicada en diversas situaciones, en conocimiento de ciertas consecuencias derivadas de su implementación.

En GeneXus se definen 'patrones activos' como aquellos patrones de *software* que es posible "instanciar", entendiéndose esto en el sentido que el marco produce todos los objetos necesarios para implementar una instancia del patrón [GCW607]. Pasan de ser una descripción escrita a una implementación directa de la solución expresada en código y operativa: *"Una patrón sistemáticamente denomina, motiva y explica un diseño general que soluciona un problema recurrente de diseño en aplicaciones GeneXus. Describe el problema, la solución y cuando aplicarla, así como sus consecuencias. Brinda también consejos de implementación y ejemplos. La solución es un arreglo general de objetos que resuelven el problema; siendo adaptada e implementada para resolver el problema en un contexto particular"* [GCW807].

6.3 Herramientas adicionales

Se ofrecen herramientas adicionales e interfases de programación, a continuación se enumeran algunas de ellas.

- **GxPublic** Expone la información de una KB, permitiendo acceder, modificar información y ejecutar servicios [Gxp105] [Gxp205].
- **GeneXus BPM Suite** Consiste en un conjunto de herramientas [GBS07] que dan soporte para cumplir con un ciclo de vida *Business Process Management* [63] (BPM) integrado por cuatro componentes bajo GxFlow [Gxf07].
- **GxQuery/Gxplorer** Permiten la elaboración dinámica y mantenimiento automático de consultas sobre bases de datos, a partir de una metadata generada con la información de las KB [GQE07] [GXP07].
- **GxPatterns** Permite la implementación de patrones, con los cuales se crearán objetos en forma automatizada [GCW907].

6.4 GeneXus X

En esta sección se expondrá acerca de la versión "GeneXus X", conocida como versión "Rocha" durante su proceso de desarrollo.

6.4.1 Descripción

La versión GeneXus X [GX08] se ha construido en base a paquetes de *software* denominados extensiones que se incorporan al *GeneXus Development Environment* (6.2.9) de manera dinámica. De esta forma el producto presenta un radical cambio de arquitectura con respecto a versiones anteriores.

[63] Administración de procesos de negocios. http://www.bmpi.org

Una característica fundamental es que dichas extensiones pueden ser producidas por terceros, lo cual abrirá un abanico muy grande de posibilidades para el agregado de funcionalidad bajo la forma de paquetes que podrán liberarse en forma comercial o gratuita. Se sostiene que no habrá diferencias, en teoría, entre los paquetes que un tercero pueda incorporar de aquellos que produzca Artech para implementar el producto. Estas extensiones se escriben en lenguaje C#, no siendo posible por el momento programarlas en GeneXus.

Según Lamas [Lam07] las posibilidades de las extensiones incluyen:

- Leer y grabar información de la KB.
- Definir nuevos objetos, partes[64] de objetos y categorías.
- Programar sobre la lógica de GeneXus[65].
- Utilizar y proveer eventos y servicios.
- Integrarse con la interfase de usuario de GeneXus.

La versión ha estado en proceso de elaboración mientras se escribía este trabajo. Previamente a su liberación final han sido liberadas versiones RC "*Release Candidate*" precedidas de versiones "*Beta"* [GCW507] y "*Community Technology Previews*" [66] (CTP), acompañadas de su correspondiente "*Software Development Kit*" [67] (SDK), bajo un programa de adopción temprana de tecnología[68] (EAP) [GCW407]. Artech abrió un foro en Internet sobre tópicos relacionados con la versión en desarrollo [Gxf06] y publicó en un sitio *web* extensiones producidas por terceros, ofreciendo en [GCW207] una lista de las disponibles. Brindó además la posibilidad de descargar a diario las nuevas versiones[69] [GRN07] del producto.

Otras novedades muy importantes en el producto la constituyen nuevos objetos como ser "*Data Selectors*" (6.4.3) , "*Data Providers*" (6.4.4), un servidor *wiki*[70] integrado, la obtención de diagramas mejorados, la integración con el diseño de flujos de trabajo (*workflow*) y con patrones activos, un IDE renovado y diferente, mejoras para el trabajo en equipo y el versionado. GeneXus X ha sido orientado hacia la generación de aplicaciones Web 2.0 [Web20].

En las figuras 6-21 y 6-22 se presenta respectivamente el aspecto del IDE y la ventana del administrador de extensiones (*extension manager*) con un conjunto de extensiones base y de terceros ya cargadas.

[64] El vocablo "partes" refiere a la estructura, reglas, documentación y otros. Se visualizan como "*tabs*" (pestañas) en el editor.
[65] Lógica de normalización y reconstrucción de tablas de la base de datos.
[66] Inspección tecnológica comunitaria anticipada http://www.gxopen.com/commwiki/servlet/hwiki?CTP.
[67] *Software Development Kit – Kit para desarrollo de software.*http://es.wikipedia.org/wiki/Software_development_kit.
[68] *Early Adopters Program.*
[69] *Night Builds*
[70] http://en.wikipedia.org/wiki/WikiWikiWeb

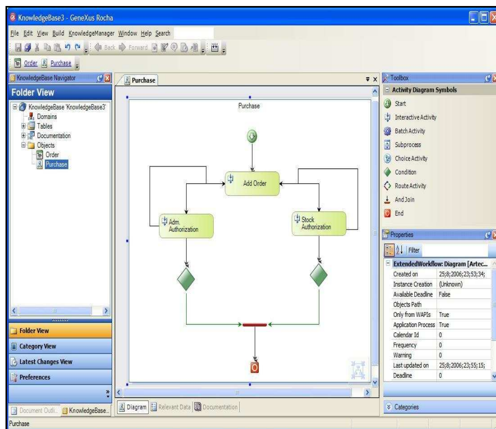


Fig. 6- 21: IDE GeneXus X -WorkFlow-

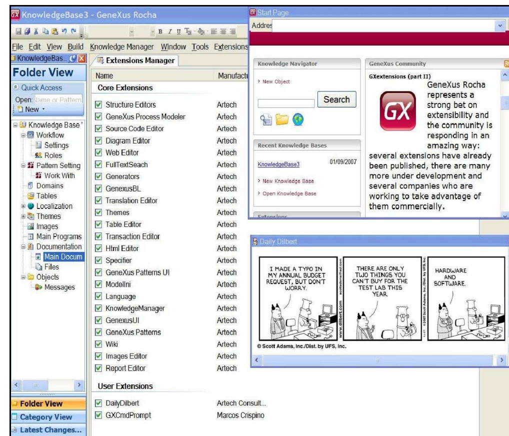


Fig. 6- 22: IDE con Extensiones cargadas

6.4.2 Arquitectura básica

El producto se construyó en base a una arquitectura en tres capas, según Lamas [Lam07], tal como se ilustra en las figuras 6-23 y 6-24 donde se muestra una visión de primer nivel de la misma; conteniendo una capa de interfase de usuario (superior), una capa de lógica (media) y una de datos (inferior).

En la capa de lógica (*Business Logic*) se destaca un componente base "Modelo de datos universal" (*Universal Data Model*), que brinda acceso a datos en forma universal a cualquier objeto; en esa capa también se encuentra el núcleo que ejecuta la lógica junto a otros componentes, representados como "Gx1" "Gx2" en las figuras 6-23 y 6-24.

En la capa de usuario reside el "UI *Framework*", encargado del esquema general del diálogo, con la ventana principal y el "*Gx Core* UI" conteniendo la interfase de usuario, con los elementos básicos siempre presentes; siendo la funcionalidad complementada con los módulos "GxUIn" que contienen editores de temas y de formularios, pudiéndose incorporar nuevos y sustituir los existentes.

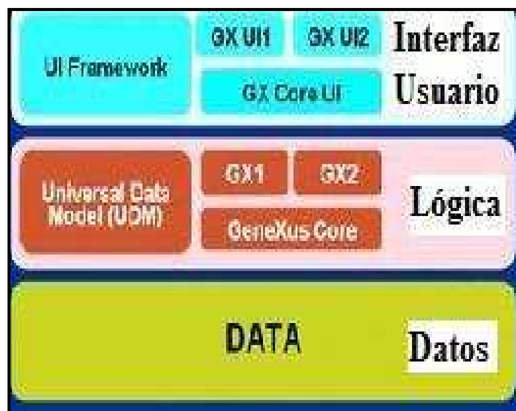


Fig. 6- 23. Arquitectura básica [Lam07]

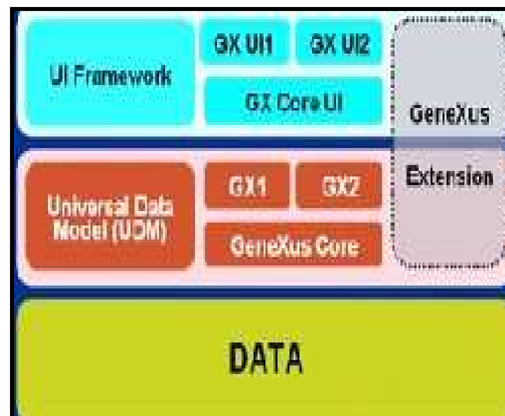


Fig. 6- 24: Arquitectura y Extensiones [Lam07]

La extensibilidad se logró en base a paquetes y servicios. Las extensiones, conocidas como "*GxExtensions*", se incorporan abarcando las dos capas superiores, lo cual se ejemplifica en la figura 6-24, aunque a futuro se indica que se permitirá ubicarlas en una única capa. En [Cri07] se ofrece una demostración de la construcción de estas extensiones.

6.4.3 *Data Selectors* (DS)

El *Data Selector* es un nuevo tipo de objeto que permite reutilizar "navegaciones", evitar la duplicación de código y otorgar una mejor forma de desacoplar la interfase de usuario de la lógica del negocio. Tiene varias secciones, donde almacena un conjunto de parámetros, varias condiciones, un orden de presentación y una estructura compuesta por atributos y formulas locales. Puede ser invocado desde otras consultas, posibilitando la reutilización. En la figura 6-25 se muestra la definición de un *Data Selector*.

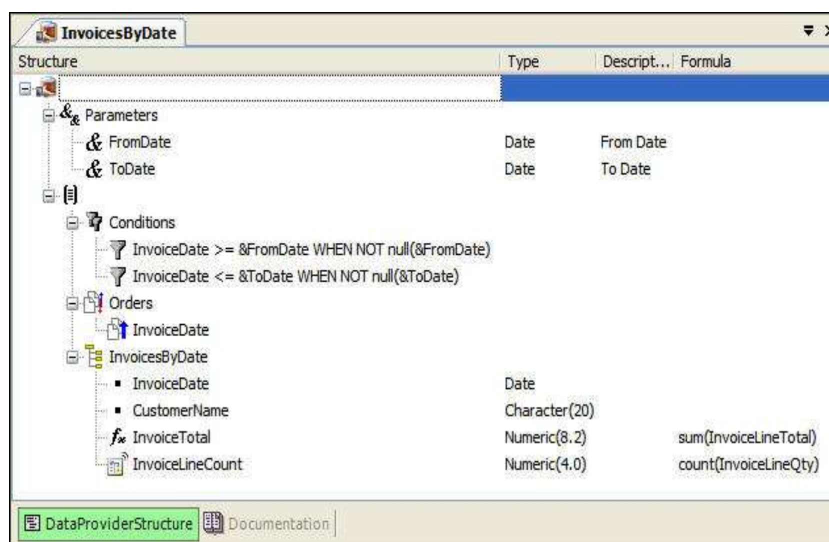


Fig. 6- 25: Estructura de un *Data Selector* [GCW707].

Ejemplo de utilización

Si se supone dada una transacción de clientes, a la cual se debe acceder desde otros objetos para obtener una lista de clientes activos identificados como tales por un atributo, basta entonces definir un DS "ClientesActivos" y usarlo en cada comando *For Each* y otros lugares donde se necesite.

En el comando *For Each*

En la figura 6-26 se ofrece la sintaxis de un *Data Selector* en un *For Each* y un ejemplo de uso referido al ejemplo expuesto en la figura 6-25.

```
Sintaxis

For Each Usign DataProviderName([parm1 [,parm2 [, ..]  ])
     ................
EndFor

Ejemplo

// El código entre For Each y EndFor se ejecuta para cada
// "tupla" devuelta por InvoicesByDate().

For Each Using InvoicesByDate(&FromDate, &ToDate)
           ...........
EndFor
```

Fig. 6- 26: Sintaxis y ejemplo de *Data Selector* en For Each [GCW707].

Es posible anidar Data Selectors.

6.4.4 *Data Providers* (DP)

Mediante los objetos *Data Providers,* GeneXus retoma aspectos metodológicos propuestos por la teoría de Warnier-Orr (6.2.1) y maximiza la aplicación de la recomendación práctica de expresar intenciones[71] en el código.

En estos objetos se define declarativamente una salida a obtener y su formato. Según se expresa en [GCW1207] pueden asimilarse a "procedimientos declarativos" que se usan para extraer información de la base de datos y otras fuentes, volcando la misma en un formato predeterminado, por ejemplo XML.

Un DP puede ser visto como un proceso que consta de entradas, transformaciones y salidas, pero que pone el foco en el lenguaje de especificación de salidas, cuya descripción se ofrece en [GCW1107]. Según expresan Fernández y Márquez estos objetos, junto a los *Business Components*, jugarán un papel preponderante en la construcción de la capa de negocios de las aplicaciones GeneXus [FM07].

En las tablas 6-1 y 6-2 se brindan ejemplos.

[71] Es posible encontrar similitudes, como en otros aspectos de GeneXus, con las ideas sobre "*intentional programming*" de Charles Simonyi [Sim95] y aquellas orientadas a la generación a partir de modelos, donde priman los aspectos declarativos.
Se alinea, en otro orden, también con la recomendación de expresar claramente intenciones en el código, dada para XP y TDD [Jef00].

Codificación	Salida Obtenida

Clients
{
Client
{
ClientId
ClientName
}
}
	<Clients>
	<Client>
	<Code>1</Code>
	<Name>JohnSmith</Name>
	</Client>
	<Client>
	<Code>2</Code>
	<Name>JenniferLopez</Name>
	</Client>
	...
	</Clientes>

Employess parm(&UserId)
{
Employee
{
Id = EmployeeId
Name = EmployeeName
EarningInfo
Where
IsAutorized(&UserId)
{
Salary = EmployeeSalary
Bonus = EmployeeBonus
}
}
}
	<Employess>
	<Employee>
	<Id>123</Id>
	<Name>John Doe</Name>
	<EarningInfo>
	<Salary>30000</Salary>
	<Bonus>5000</Bonus>
	</EarningInfo>
	</Employee>
	...
	</Employees>

Tabla 6- 1 Ejemplos de Data Providers [GCW1107] [GCW1207].

```
&CustomerId = 1
&Tabs = Tabs(&CustomerId)
// Nota: &Tabs es un SDT, Tabs() es un DataProvider
```

Tabla 6- 2 Ejemplos de utilización de *Data Providers* [GCW1007].

La salida de un DP puede ser asignada a un SDT o a un BC.

6.4.5 Ciclo de vida y base de conocimiento

El concepto de modelos desaparece dando lugar a un ambiente de desarrollo más integrado, donde durante el ciclo de vida no es necesario estar cambiando al modelo de diseño (6.2.5) para agregar o cambiar las visiones de los usuarios. Se definen además "*enviroments*" para los cuales se genera, una KB puede definirse con varios "*environments*" (por ejemplo Java o .Net).

6.4.6 Administración de versiones

Durante el ciclo de vida se alcanzan determinados hitos importantes para las personas que trabajan en el proyecto. Estos hitos son denominados "*versions*"

(versiones) y consisten en imágenes de solo lectura de la aplicación en un momento dado. Por otra parte, mientras algunos desarrolladores trabajan en la próxima versión de una aplicación otros están efectuando mantenimiento en la versión actual y en general se desea mantener hilos separados de cambio, que se denominan "*branches*" (ramas). Una KB puede almacenar tantas versiones y ramas como sea necesario mantener. GeneXus X provee de un mecanismo para su administración [GCW108].

6.5 Consideraciones

El desarrollo de *software* utilizando GeneXus brinda múltiples características ventajosas. En particular, GeneXus sintetiza las visiones de los usuarios para construir y mantener las bases de datos, posibilita la aplicación práctica de una filosofía de desarrollo incremental e iterativo, mantiene automáticamente las aplicaciones y admite la inclusión de objetos del usuario, brinda independencia tecnológica, permite la aplicación de patrones activos y es extensible. Presenta debilidades tales como la carencia de funcionalidades integradas y específicas para la prueba unitaria y la dificultad de aplicación de disciplinas basadas en TDD. Pueden estimarse como riesgos el hecho de ser una herramienta propietaria y la dependencia que se genera acerca de la futura selección de tecnologías para las cuales generará aplicaciones.

Más allá de las debilidades y riesgos enumerados, ha sido posible migrar aplicaciones construidas con GeneXus entre ambientes disímiles a costo reducido, incluso hacia arquitecturas totalmente desconocidas al momento de haber creado las aplicaciones originales y se han logrado importantes incrementos de productividad.

La lista de lenguajes, plataformas y bases de datos soportadas ha acompañado la evolución de la tecnología. Empresas del medio local lograron insertarse en el mercado internacional con productos desarrollados con la herramienta, la empresa creadora adquirió alcance y difusión internacional, siendo muy alta la participación e involucramiento de la comunidad de desarrolladores.

Por último, se entiende pertinente destacar que en el ámbito académico se han desarrollado varias investigaciones, proyectos y tesis, relacionadas de alguna forma con la herramienta o su uso, entre otros [Cas07], [LSLN03], [Per04], [Sal06] y el proyecto descrito en (A).

Capítulo 7

PROPUESTA DE ESPECIFICACIÓN

En este capítulo se presenta la especificación de requerimientos para una herramienta de pruebas unitarias asociada a GeneXus (6), que recoge funcionalidades del *Framework for Integrated Tests* (FIT) (5).

Está organizado de la siguiente manera:

- En la sección 7.1 se brinda una introducción, la visión, el objetivo general, así como una exposición sumaria de la plataforma tecnológica.
- En la sección 7.2 se enuncian hipótesis de trabajo y restricciones.
- En la sección 7.3 se explicitan los objetivos específicos.
- En la sección 7.4 se ofrece el documento de requerimientos.
- En la sección 7.5 se expone el modelo de dominio.

El formato de la presentación, en las secciones 7.1 a 7.3, recoge recomendaciones extraídas de [BV02].

7.1 Introducción

En esta sección se brindan antecedentes, una introducción, la visión, el objetivo general y una exposición sumaria de la plataforma tecnológica.

7.1.1 Antecedentes

En el año 2003 Enrique Almeida anunció su objetivo de crear una herramienta de automatización de pruebas unitarias en GeneXus, a la que denominó GXUnit. En el año 2004 formalizó su propuesta [Alm04] caracterizando la misma en la línea de las herramientas xUnit; haciendo un llamado para conseguir adeptos y formar equipos para su desarrollo, bajo una modalidad de código libre. Almeida planteó que los objetivos básicos a cumplir serían los siguientes:

- Crear un marco de pruebas asociado a GeneXus.
- Poder escribir las pruebas en GeneXus, bajo la forma de *procedures.*
- Ejecutar las pruebas y registrar los resultados.

Considerando para el desarrollo un enfoque iterativo e incremental postuló ir cumpliendo con el objetivo de generar objetos[72] para prueba de:

- Objetos sin UI (*Procedures*)
- *WebPanels*
- Transacciones (*Transactions*)
- *WorkPanels*

Por último, Almeida planteó la aspiración de contar en GeneXus con nuevos comandos[73], mejorar el manejo de excepciones, definir objetos de prueba y la capacidad de integrar la herramienta a su IDE.

En 2006 se retoma bajo la forma de "Proyecto Colaborativo"[74] al cual nos integramos junto con Uruguay Larre Borges. Se definieron como objetivos elaborar una especificación funcional preliminar y estudiar opciones para viabilizar su programación, definiéndose los roles de cada uno de los tres participantes del proyecto. En [Gxu206] y [Gxu106] se expuso acerca de la experiencia y se publicó una aproximación a su especificación funcional.

En agosto de 2007 dos grupos de estudiantes del curso "Proyecto de Ingeniería de *Software*" de la Facultad de Ingeniería comenzaron a desarrollar GXUnit en el contexto del mencionado curso. Se acordó con el resto de los proponentes del proyecto los requerimientos mínimos a cumplir, incluyéndose funcionalidad relativa

[72] La palabra "objeto" se refiere en el contexto GeneXus a "instancias de objetos tipos GeneXus".
[73] Try/Catch y Assert.
[74] Proyectos orientados a la comunidad de desarrolladores GeneXus con la participación de varios miembros de la misma, de duración breve.

a la parametrización de las pruebas mediante datos externos en formato tabular, propuesta en la contribución principal de este trabajo, como un requerimiento básico para la definición de las pruebas; oficiando de prueba de concepto. Otro importante requerimiento común es la generación de código GeneXus para pruebas, de forma de permitir la portabilidad. A partir de ese momento se actúa en el rol de Cliente, realizándose el seguimiento de la construcción y validación de los prototipos, junto al resto de los proponentes. En el ANEXO A se brindará información detallada sobre las dos versiones de GXUnit [Gxu108] [Gxu208].

7.1.2 Presentación

La presente propuesta consiste en la especificación de una herramienta de pruebas unitarias asociada a GeneXus que incorpora funcionalidades inspiradas en FIT.

¿Para qué se propone?

La propuesta apunta a facilitar las pruebas unitarias, por la vía de brindar soporte para la especificación de los casos de prueba y su automatización; concibiéndose asociada a GXUnit. Intenta enriquecer la propuesta original de GXUnit con nuevos mecanismos de definición, ejecución y visualización de las pruebas.

Surge como respuesta a las preguntas:

- ¿Qué requerimientos debería cumplir el marco de pruebas unitarias para aprovechar las características de GeneXus en la elaboración de los programas para prueba y el mantenimiento de los casos de prueba en forma automatizada?
- ¿Cómo adaptar funcionalidades inspiradas en FIT para dicho marco de pruebas unitarias de programas producidos por GeneXus teniendo en cuenta sus particularidades, posibilidades y características?

Se la concibe como un complemento funcional de GXUnit, denominado GXFIT, y se plantea como parte del conjunto de herramientas que podrían producirse para incorporar las prestaciones ausentes en GeneXus. Comparte un núcleo de requerimientos básicos con GXUnit que se enuncian respondiendo a la primera interrogante, agregando aquellos que permitan llegar a responder la segunda.

Visión

"El propósito es brindar al Analista[75] GeneXus un mecanismo para automatizar la creación, mantenimiento y ejecución de pruebas unitarias basado en funcionalidades adaptadas desde FIT".

[75] Se utilizará el termino "Analista" para referirse a los desarrolladores que utilizan GeneXus.

¿Por qué la referencia a una adaptación?

El término adapta adquiere especial importancia en este contexto, pues, entre otras razones:

- FIT es una herramienta diseñada para pruebas de aceptación. El alcance que se dará a la presente propuesta se limita a pruebas unitarias.
- Las implementaciones de FIT para otros lenguajes diferentes de Java se han realizado básicamente traduciendo el código Java al nuevo lenguaje. Dado en este caso que no tiene sentido considerar portar FIT traduciendo el código original se resuelve referirse a una adaptación de funcionalidad inspirada en FIT que complemente la herramienta de pruebas GXUnit[76].
- Una implementación cualquiera de FIT brinda un ejecutable (*FileRunner*) (5.2) para correr las pruebas en el lenguaje y plataforma al que sido portado. Dicho programa interpreta información suministrada en tablas, desde donde determina el tipo de *fixture* (5.2.1) a utilizar, dispara los métodos adecuados y busca en los ejecutables y bibliotecas compiladas los métodos de prueba para correrlos, entregando los resultados también con formato de tablas análogos a los utilizados para contener la definición de las pruebas. Los imprevistos disparan excepciones que son capturadas por el marco, el cual notifica y sigue ejecutando. La propuesta propone un comportamiento diferente: los objetos[77] GeneXus que implementen el análogo a las *fixtures* contendrán código GeneXus y se generará con ellos programas para correr las pruebas en el lenguaje de destino. La creación y el mantenimiento del código de implementación de dichos objetos será automatizado, lo que apunta a resolver el problema del mantenimiento de *fixtures* detectado en FIT en base a las prestaciones de GeneXus.
- Para utilizar FIT el desarrollador escribe clases que proveen métodos que funcionan como adaptadores entre los casos de prueba descritos en las tablas y los métodos a ejecutar de las clases del SUT (5.2.1). Esto le da un grado de independencia importante con relación a los objetos a probar. Los adaptadores se apoyan en las *fixtures* entregadas por el marco, facilitando la escritura, ocultando la complejidad y prescribiendo comportamiento según la *fixture* a aplicar. Al especificarse los casos de prueba se propone hacer abstracción de los objetos del SUT y utilizar ATDD (3.3.1). En cambio, con la presente propuesta, el énfasis estará puesto en la creación de los objetos que permitirán ejecutar las pruebas a partir de objetos del SUT existentes en la KB.
- Se propondrá que la herramienta deba forzar y mantener un formato adecuado de las tablas en relación directa con los objetos a probar y los programas de prueba, permitiendo la reconstrucción automatizada de dichas tablas. Esto se alinea con las facilidades en las cuales destaca GeneXus y

[76] Por otra parte, aunque la portabilidad anteriormente descartada fuera posible, FIT es una herramienta liberada bajo licencia GNU-GPL, en tanto que GeneXus es propietaria.

[77] Se utilizará en el presente capítulo la palabra “objeto” como sinónimo de “objeto GeneXus” (6.2.9). No se refiere a objetos tal como se les conoce en OOLP.

apunta a resolver el problema planteado en FIT por el mantenimiento de las tablas con los casos de pruebas, ante cambios en los objetos a probar.

La herramienta deberá concebirse de forma lo suficientemente modular como para permitir agregarle componentes a futuro.

7.1.3 Objetivos

Los objetivos de la propuesta consisten en especificar una implementación que:

- Permita crear y mantener en forma automatizada programas especializados en pruebas unitarias parametrizables, de programas producidos a partir de objetos GeneXus, que implementen funcionalidades inspiradas en el *FrameWork for Integrated Tests.*
- Brinde facilidades para el ingreso y mantenimiento de los casos de prueba, el almacenamiento y mantenimiento de los archivos con parámetros (acciones, datos de entrada y resultados esperados), bitácora con resultados obtenidos y visualización de resultados, a partir de la información disponible en la KB del SUT.

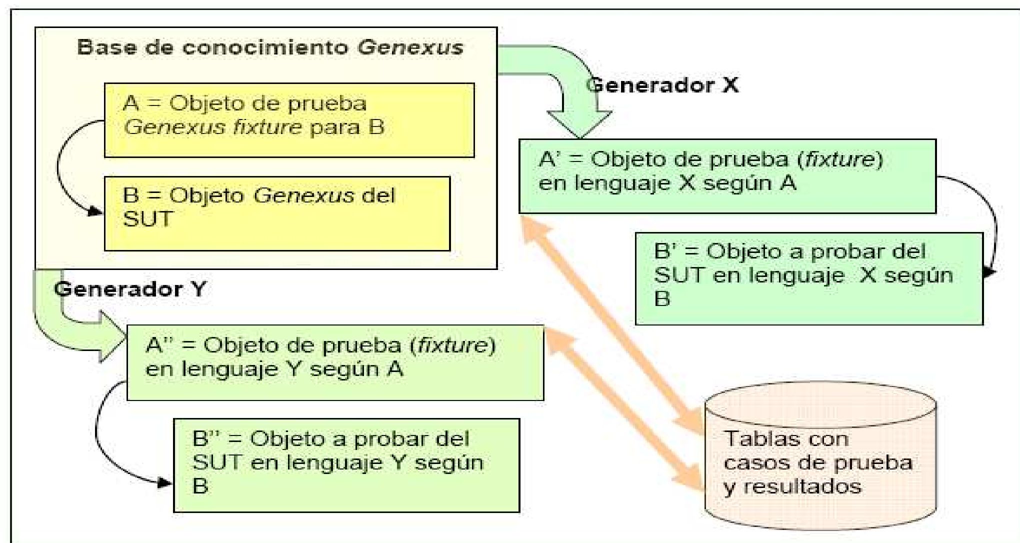


Fig. 7- 1: Objetos para prueba.

El objetivo de la presente propuesta es producir un conjunto de artefactos que sirvan como base para la elaboración de la funcionalidad planteada.

Artefactos a producir

El presente trabajo produce como entregables los artefactos que se enumeran a continuación:

- Documento de requerimientos y especificación funcional.
- Modelo del dominio.

7.1.4 Plataforma tecnológica

Se establece que la herramienta se desarrolla para GeneXus X, donde se cumplen los preceptos que se enumeran a continuación, según se documenta en [GCW107] y se describió en (6.4):

- Esta desarrollada en base a extensiones, por lo que el propio producto puede ser visto como un conjunto de extensiones adicionadas al *GeneXus Development Environment* (6.2.9).
- Se permite la programación de extensiones, bajo la forma de paquetes, que se agregan al *GeneXus Development Environment*'.
- Las extensiones potencian las posibilidades de interacción con el IDE y las KB, por ejemplo permitiendo crear nuevos tipos de objetos a los cuales especificarles propiedades y editores.

7.2 Hipótesis y restricciones

En esta sección se enuncian hipótesis y restricciones referidas a la presente propuesta.

7.2.1 Supuestos

Se plantearon como válidos los siguientes enunciados:

GeneXus:

- Durante la elaboración de este trabajo de tesis la versión actual GeneXus X, se encontraba en desarrollo. Se estableció entonces como supuesto básico que la misma se efectivizaría como la nueva versión de GeneXus; permitiendo adicionar extensiones según consta en [GCW107].

GXFIT:

Dadas las posibilidades de las extensiones podrá:

- Utilizar servicios de GXUnit o integrarse como parte del código de dicha extensión.
- Acceder a la información mínima necesaria acerca de los objetos a probar y sus relacionados de la KB del SUT.
- Comandar acciones, tales como crear, modificar y destruir objetos.
- Incorporar a GeneXus nuevos tipos de objetos o partes de objetos, con propiedades, editores asociados, métodos; utilizables como si fueran objetos nativos.

7.2.2 Restricciones

Se imponen las siguientes restricciones:

- Dominio de objetos 'verificables':
  - Permite la prueba de los programas generados a partir de los siguientes objetos: *Procedures* (6.2.10.4) y *Transactions Business Component* (BC) (6.2.10.2), validación de datos entregados por las navegaciones de los *Data Selectors* (DS) (6.4.3), *Data Providers* (DP) (6.4.4) y datos contenidos en tablas de la base de datos.
  - Los programas a probar no utilizarán interfase de usuario (UI).
  - No se validarán reportes.
  - Se podrá establecer en la especificación restricciones adicionales, por ejemplo: según los tipos de datos o la persistencia (*commit*).
- No utilizará código fuente de FIT ni de ninguna de sus traducciones. Tampoco ensamblará ejecutables de dicha procedencia.

7.3 Alcance

En esta sección se especifica el alcance de la propuesta, enumerando y describiendo tanto los objetivos específicos como los entregables que deberán producirse.

7.3.1 Objetivos específicos

7.3.1.1 Objetos para prueba

Enunciar un conjunto de requerimientos que especifiquen una implementación para:

- Automatizar la generación y mantenimiento de programas verificadores, especializados en ejecutar pruebas unitarias parametrizables, elaborados a partir de los objetos GeneXus definidos como "verificables".
- Crear y mantener de forma automatizada las tablas que contendrán los parámetros para las pruebas.

7.3.1.2 Adaptación de *fixtures* a GeneXus

Enunciar un conjunto de requerimientos que especifiquen las funcionalidades de prueba a implementar.

Se decide que en la primera versión, límite para este trabajo, se utilicen conceptos de las siguientes *fixtures*, descritas en (5.2.1):

- *Action Fixture*
- *Column Fixture*

- *Row Fixture*
- *Set Up Fixture*

La decisión se justifica dado que las tres primeras son las *fixture*s básicas de FIT, alineándose su implementación con el enunciado del primer objetivo del proyecto FIT [Cun107], y la última pues corresponde a la inicialización de datos.

7.3.1.3 Formatos de las tablas

Describir los formatos de las tablas que almacenarán los parámetros.

7.3.1.4 Persistencia de los datos

Enunciar requerimientos sobre la persistencia de:

- Los parámetros para las pruebas.
- Los resultados de las pruebas.

7.3.1.5 Editor de las tablas y casos de pruebas

Enunciar requerimientos para un editor de tablas y casos de pruebas que permita:

- Crear y editar las tablas con los parámetros para las pruebas.
- Crear y editar los casos de prueba.
- Editar documentación asociada.
- Vincular los casos de prueba y las tablas de parámetros con los objetos a probar y los programas de prueba.

7.3.1.6 Resultados de las pruebas

Enunciar requerimientos acerca de:

- El formato de presentación de los resultados de las pruebas.
- El módulo de presentación de los resultados de las pruebas.

Entregable:

El cumplimiento de los objetivos específicos anteriormente expuestos se asocia a la producción de un documento de requerimientos y especificación funcional.

7.3.1.7 Modelo de dominio

Construir un modelo de dominio o conceptual.

Este tipo de diagrama expresa el resultado de la descomposición del dominio del problema en conceptos, atributos y relaciones entre ellos; ilustrando acerca de las clases conceptuales (ideas, cosas, objetos) mediante diagramas. Este modelo permite brindar una abstracción, desplegando una vista que ignore los detalles innecesarios [Lar04].

Entregable:

El cumplimiento del objetivo específico anteriormente expuesto se asocia a la producción de un diagrama de modelo de dominio.

7.4 Requerimientos

Los requerimientos [78] se expresarán mediante una visión del producto más un conjunto de relatos[79] (*stories*) siguiendo las recomendaciones metodológicas de XP (que se resumen en el Anexo B a este trabajo). Se ofrecerá también una breve descripción funcional.

Para mayor claridad en la exposición se referirá a los programas que se ejecutan en las pruebas mencionando al tipo de objeto GeneXus que los genera, por ejemplo: "ejecución de un procedimiento (*procedure*)" o "probar un procedimiento (*procedure*)".

7.4.1 Visión

Se comenzará exponiendo la visión sobre el propósito de creación del producto, expresada en forma concisa:

"El propósito es brindar al Analista GeneXus un mecanismo para automatizar la creación, mantenimiento y ejecución de pruebas unitarias basado en funcionalidades adaptadas desde FIT".

7.4.2 Descripción funcional

A continuación se brinda una breve descripción funcional general[80]

"El Analista utilizará la herramienta desde el IDE de GeneXus para definir y ejecutar casos de prueba correspondientes a pruebas unitarias. Contará también con la posibilidad de ejecutar las mismas en modo 'desatendido' (por lotes).

[78] Se utilizará la palabra "requerimientos" dada su aceptación y uso generalizado en la Ingeniería de Software, a pesar del comentario realizado en [BA04] acerca de su connotación inhibidora de la actitud de abrazar el cambio propuesta por XP.
[79] Se utilizará la palabra relatos como traducción de *stories* dentro de este trabajo de tesis.
[80] En el curso de esta descripción se hará referencia a conceptos desarrollados en (5) y (6) . Su inclusión se realiza con el objetivo de facilitar la creación de la metáfora.

A partir de los casos de prueba se crearán y mantendrán en forma automatizada programas especializados para probar *Business Components*, *Data Providers, Data Selectors, Procedures* y para validación del contenido de la base de datos. Los programas a probar no podrán requerir UI ni invocarán a otros que la requieran al momento de su prueba.

Los programas especializados para prueba serán generados a partir de objetos GeneXus, para asegurar la portabilidad, adaptando funcionalidades de prueba propuestas por FIT (*Framework for Integrated Test*). En particular se utilizará un algoritmo similar al propuesto por *ColumnFixture* para probar *Procedures*, *ActiónFixture* para probar *Business Components* y *RowFixture* para *DataProviders*, *Data Selectors* y contenido de la base de datos.

Los casos de prueba que el Analista podrá definir serán parametrizables, lo cual significa que la herramienta se enmarcará en el conjunto de herramientas para prueba guiadas por datos y comandos. Dichos parámetros (datos, palabras claves, comandos, resultados) residirán en tablas con formato predeterminado que los programas para prueba interpretarán. La herramienta creará dichas tablas imponiendo el formato adecuado según la funcionalidad prevista a desplegar por la prueba, brindando editores especializados para que el Analista introduzca los valores.

A los efectos de conseguir cierto grado de aislamiento de los objetos a probar se prevé que el Analista tenga facilidades para sustituir los objetos colaboradores con dobles para pruebas, a los que pueda especificar respuestas premeditadas a ciertos estímulos.

La herramienta se encargará de reconstruir, en los casos en que sea posible hacerlo, las tablas con datos de prueba y los objetos para prueba al detectar ciertos cambios en los objetos a probar.

El Analista podrá seleccionar que pruebas ejecutar. Obtendrá una visión general de cuales pasaron, cuales fallaron, cuales fueron ejecutadas parcialmente y las excepciones, siguiendo la convención de colores que expresa la siguiente semántica: rojo=falla, verde=pasa, gris=parcialmente ejecutada, amarillo=excepción.

Obtendrá también un informe del resultado de cada prueba, que se mostrará en formato análogo al usado para presentar los datos, agregando los valores obtenidos junto a los esperados, con los resultados destacados según la convención ya establecida. Se establecerán restricciones al tipo de validaciones automáticas que se podrán realizar y los tipos de datos y salidas a considerar en las mismas.

El Analista podrá visualizar una historia de las pruebas anteriormente realizadas y su resultado, así como también saber que objetos a probar pasaron las pruebas".

7.4.3 Características técnicas generales

Extensión
Los Analistas utilizarán los servicios de la herramienta desde el IDE de GeneXus X. Se construirá esta herramienta bajo la forma de una o varias extensiones para dicha versión de GeneXus. Se deberá programar en C# utilizando el SDK suministrado para la construcción de extensiones.

Diseño
El diseño de la herramienta deberá ser modular. A futuro se espera poder incorporarle componentes que permitan generar pruebas para otros objetos GeneXus no contemplados en el dominio inicial de objetos a probar, así como nuevos editores para los casos de prueba. El reuso y la independencia entre capas se consideran aspectos muy importantes a ser tenidos en cuenta.

Documentación
La herramienta se desarrollará en incrementos, por parte de equipos diferentes de desarrolladores. Será necesario entonces que cada equipo mantenga y publique notas para facilitar la continuación del desarrollo a futuros equipos.

7.4.4 Relatos

Se exponen a continuación un conjunto de relatos que contienen los requerimientos. Siguiendo las recomendaciones para la confección de los relatos dada por Jeffries et al. en [Jef00] y Beck y Andres en [BA04] se abordarán solamente los aspectos que se consideren esenciales.

Acerca de objetos y parámetros

Royp1. Dominio de los objetos a probar
Los objetos GeneXus para los que el Analista podrá crear, mantener y efectuar pruebas unitarias se identifican como “verificables”. Dichos objetos son:

- *Business Components*
- *Data Providers*
- *Data Selectors*
- *Procedures (procedimientos)* que no requieran interfase de usuario (UI) ni invoquen a otros objetos que si la requieran.

Royp2. Casos de prueba (TC)
El Analista creará “casos de prueba” (TC) (***test case***) para especificar las pruebas de objetos verificables. La extensión implementará lo necesario para definir y contener los casos de pruebas., a los cuales les asignará una propiedad: ***test***.

Royp3. Objetos para implementar pruebas *(test*)
La herramienta generará y mantendrá en forma automatizada objetos GeneXus para implementar pruebas unitarias a partir de las especificaciones contenidas en los TC. Dichos objetos deberán poseer, además de la propiedad ***test***, una propiedad que los identifique como automáticamente generados (***testaut)***.

Royp4. Modificación manual de *testaut*
Se debe impedir la modificación manual desde el IDE de los objetos identificados con la propiedad ***testaut.*** Estos serán generados y mantenidos por la herramienta.

Royp5. Parámetros para pruebas
Las pruebas se parametrizan. El Analista suministrará dichos parámetros. Los parámetros corresponden, como mínimo, a datos de entrada, resultados esperados y comandos. Dichos datos, resultados esperados y comandos se organizan en formato tabular. La extensión suministrará editores apropiados para los diferentes tipos de tablas.

Royp6. Almacenamiento de parámetros para las pruebas
Los parámetros para las pruebas se podrán almacenar en la base de conocimiento (KB) GeneXus del SUT o en forma externa, accesibles para la extensión. De optarse por almacenamiento externo, deberá ser posible su total y completa trazabilidad desde y hacia los TC donde intervienen.

Royp7. Creación de un TC
El Analista podrá indicar la creación de un TC para cualquier objeto verificable. Identificará el TC, brindará su descripción e ingresará los parámetros.

Royp8. Edición de un TC
El Analista podrá editar un TC ya existente, cambiando su descripción y los datos contenidos en sus parámetros.

Royp9. Eliminación de un TC
El Analista podrá eliminar un TC existente. La acción de eliminar un TC provoca la eliminación en cascada de todos los objetos para pruebas (*testaut*) generados a partir del TC y de los datos para las pruebas (parámetros) asociados al mismo.

Royp10. Comentarios para un TC
El Analista podrá ingresar y modificar comentarios generales asociados a un TC, para documentar la prueba. Podrá también ingresar y modificar comentarios asociados a cada ejecución de una prueba.

Royp11. Ejecución de las pruebas (*test*)
Los programas para ejecutar las pruebas unitarias serán especificados, generados y ejecutados desde el IDE de GeneXus. El Analista desea tener la posibilidad de ejecutar las pruebas también en modo desatendido (por ejemplo durante un *build*). El Analista escogerá, de una lista de TCs, cuales ejecutar en la próxima prueba.

Royp12. Oráculo: automatización
El Analista necesita que se implemente un oráculo que le notifique si una prueba falló o pasó. Este oráculo debe funcionar en base a la comparación de resultados esperados ofrecidos por el Analista frente a resultados obtenidos. La comparación deberá permitir diversos operadores más allá de la igualdad.

Royp13. Oráculo: procedimientos de verificación del usuario (PVUs)
Existirán *procedures* GeneXus que se conocerán como “procedimientos verificadores del usuario” (**PVU**), que escribirá el Analista y que podrán ser asociados a los casos de prueba (TC) para ser invocados previo a la finalización de la prueba. Estos procedimientos implementarán oráculos adicionales y poseerán un único parámetro “*booleano*” para indicar si la prueba fue exitosa o fallida. El resultado obtenido se considera independiente del que se obtiene con el oráculo automático, por lo cual basta que uno cualquiera de los oráculos indique fallo para que la prueba total se considere fallida. Tendrán como mínimo un parámetro de salida y los mismos parámetros de entrada que el objeto GeneXus a probar.

Royp14. Procedimientos de inicialización del usuario (PIUs)
El Analista podrá suministrar *procedures* que se disparen por única vez al comienzo de la ejecución de un TC. Estos *procedures* serán conocidos como “procedimientos de inicialización del usuario” (**PIU**). Devolverán un parámetro de tipo “*booleano*” para indicar si finalizaron satisfactoriamente. El Analista los podrá utilizar, entre otros fines, para inicializar el ambiente, por ejemplo poblando tablas de la base de datos (BD) con datos leídos desde fuentes externas o salvando su estado previo a la prueba.

Royp15. Procedimientos de finalización del usuario (PFUs)
El Analista podrá suministrar *procedures* que se ejecuten por única vez al finalizar la ejecución de un TC. Estos *procedures* serán conocidos como “procedimientos de finalización del usuario” (**PFU**). Devolverán un parámetro de tipo “*booleano*” para indicar si finalizaron satisfactoriamente. Una de sus funciones podrá ser la de restaurar el ambiente a un estado previo a la prueba.

Royp16. Eliminación de PVUs, PIUs y PFUs
Cuando el Analista elimine un procedimiento PVU, PIU o PFU se eliminará automáticamente la referencia hacia el mismo en los TC que lo utilizan. Posteriormente se generarán nuevamente los objetos para prueba (*testaut*) para reflejar estos cambios en los TCs.

Royp17. Selección de PVUs. PIUs y PFUs a ejecutar en próxima prueba
El Analista, previo a la ejecución de un TC, podrá seleccionar los PVUs, PIUs y PFUs a ejecutar en esa corrida. Podrá indicar al TC que siempre recuerde la última selección realizada.

Royp18. Tipos de datos
El Analista suministrará parámetros según los tipos de datos básicos de GeneXus y tipos de datos estructurados (SDT). Se conocerán como los "tipos de datos reconocidos" por la herramienta. Una instancia de un tipo de objeto verificable quedará excluida del dominio de objetos a probar si necesita parámetros de entrada cuyo tipo no es contemplado por la herramienta. Así mismo, los datos de salida de tipos no reconocidos no se verificarán de forma automática, no se pedirá por lo tanto el ingreso correspondiente de los resultados esperados ni es necesario presentarlos en los informes de resultados. Se pretende que se pueda ampliar el dominio de los 'tipos de datos reconocidos' en las sucesivas versiones de la herramienta.

Acerca de la interfase general de usuario

Rius1. Semántica de colores
Se expresarán la comprobación de los resultados con una semántica de colores:
Rojo=falló Verde=Pasó Amarillo=Excepción Gris=No se ejecutó o se ejecutó parcialmente.

Rius2.Visualización de parámetros
El Analista, indicando uno de los TCs, podrá visualizar sus parámetros, su descripción y comentarios generales; así como otra información de interés: fecha y hora de la última ejecución de la prueba correspondiente y si pasó o falló. Si hubiera parámetros contenidos en archivos externos a la KB visualizará el camino y el nombre del archivo que los contiene, pudiendo abrir el mismo de contar con un intérprete de visualización según el formato de dichos archivos.

Rius3. Informe general
El Analista necesitará un informe rápido que le indique que objetos verificables del SUT pasaron todas sus pruebas, cuales no han sido probados, y cuales no pasaron. También necesitará conocer que objetos verificables no tienen TC asociados. Desea que la presentación de estos resultados luzca similar a la utilizada por los marcos xUnit, con una estructura arborescente de objetos verificables y casos de prueba asociados.

Rius4. Informe particular del resultado de una prueba actual
El Analista, señalando uno de los TCs, necesitará rápidamente visualizar los resultados de la última prueba ejecutada referida a ese TC. Se le mostrarán los resultados esperados junto a los obtenidos, siguiendo la "semántica de colores" y respetando en lo posible el mismo formato tabular en que fueron presentados los datos para la prueba.

Acerca de la bitácora

Rbit1. Almacenamiento de resultados de una prueba (bitácora)
El Analista podrá escoger si almacena en una bitácora los resultados de la ejecución de los TC, pudiendo decidir diferentes niveles de detalle en que se registrará en la

bitácora. Necesitará como mínimo, en caso de habilitarse la bitácora, que se almacene la identificación del TC y objeto probado, incluyendo día y hora de su ejecución y si pasó o falló en la prueba.

Rbit2.Visualización de resultados de ejecución de pruebas anteriores
El Analista deseará visualizar los resultados almacenados en la bitácora relativos a las ejecuciones detalladas anteriores de un TC, de haber persistido dicha información. Se le presentarán ordenados cronológicamente, dado un rango de fechas y un TC o un objeto del SUT sometido a verificación. El formato de visualización será análogo al utilizado para la visualización de resultados actuales.

Rbit3. Comparación con resultados de ejecuciones anteriores de las pruebas
El Analista podrá comparar los resultados actuales de la ejecución de los TCs con resultados anteriores (de haber persistido dicha información), dado un TC u objetos verificables del SUT. Deseará rápidamente saber cuales pruebas han tenido resultados diferentes a los obtenidos en ejecuciones anteriores.

Rbit4. Acceso a la bitácora por parte de los PVUs
Los PVUs podrán utilizar un servicio de la extensión que les permita depositar datos acerca de su corrida en el formato adecuado de la bitácora, adicionales a los datos que registra el proceso de verificación automatizado.

Rbit5. Eliminación de entradas en la bitácora.
Cuando el Analista elimine un TC podrá indicar si desea eliminar de la bitácora las entradas correspondientes a la historia de sus ejecuciones.

Persistencia

Rper1. Estado inalterado
La extensión deberá permitir incluir toda la prueba en una misma UTL de forma de implementar un *rollback* al finalizar la ejecución.

Rper2. Base de datos en memoria
El Analista desea tener la posibilidad de generar toda la aplicación del SUT y sus pruebas para una base de datos en memoria, de forma de realizar rápidamente la carga de datos y las pruebas, en forma local. Esto dependerá de la implementación de un generador para una base de datos de este tipo.

Rper3. Verificar estados
Al Analista le interesará verificar el estado de la base de datos luego de la ejecución de una prueba, por lo cual se desea proveer un mecanismo en tal sentido que le permita conocer la variación ∩ BD = BDinicial - BDfinal y verificar que el estado final sea el esperado.

Acerca de las fixtures

Rfix1. Categorización de pruebas
Las pruebas unitarias implementarán diferentes algoritmos para comandar las mismas, definiéndose los siguientes:

- "*ActionTest*" (AT).
- "*ColumnTest*" (CT).
- "*RowTest*" (RT).
- "*SetupTest*" (ST).

La clasificación mencionada no esta cerrada a futuras incorporaciones de otros algoritmos. La implementación de estos algoritmos divide el dominio de los objetos verificadores automáticos *testaut* en "categorías" llamadas "***fixtures***" .

Rfix2. Dominio
Las *fixtures* deberán permitir, dentro del dominio de objetos verificables:

- Probar *Business Components* y *Procedures.*
- Validar datos entregados por navegaciones de *Data Selectors.*
- Validar datos entregados por *Data Providers.*

También permitirán:

- Validar datos contenidos en la base de datos.
- Incorporar datos en la base de datos.

Rfix3. Correspondencia
La *fixture* principal para una prueba queda determinada según el objeto verificable a probar o datos a comprobar, según la siguiente tabla:

Objeto	***Fixture***
Business Component "BC"	*ActionTest* "AT"
DataSelector/DataProvider "DS/DP"	*RowTest* "RT" y *ColumnTest* "CT"
Procedure con parámetros	*ColumnTest* "CT"
Procedure sin parámetros	*ColumnTest* "CT"
Base de datos "BD"	*SetupTest* "ST" *RowTest* "RT"

Tabla 7- 1: Correspondencia entre objetos a probar y algoritmos.

Rfix4. Componentes (partes) de un caso de prueba
El Analista, al definir un caso de prueba, indicará:

- Un objeto verificable de uno de los siguientes tipos:
  - BC, DS, DP o *Procedure*.
- Una descripción de la prueba.
- La inicialización para la prueba (opcional)
  - Uno o varios PUIs.
  - Una o varias tablas ST (para inicializar tablas de la base de datos).

- Las *fixtures* que dirigen la prueba según la siguiente relación:
  - Para BC: AT.
  - Para DS/DP: RT y CT.
  - Para *Procedures*: CT.
- Los parámetros para la prueba (incluyendo opcionalmente vínculos a fuentes externas conteniendo los mismos)
- La verificación de la BD mediante "RT" (opcional).
- La verificación de usuario mediante uno o varios PVUs (opcional).
- La finalización de parte del usuario mediante uno o varios PFUs (opcional).
- Definiciones para *Mocks* a utilizar.

No se considera que el conjunto de componentes para un TC sea cerrado con respecto a la enumeración precedente.

Rfix5. Formato de tablas
Cada *fixture* utiliza un formato de tabla determinado para los parámetros de la prueba que implementa. El formato de tabla de cada *fixture* se conoce por el mismo nombre que su *fixture*.

Acerca del impacto de los cambios

Rimp1. Impacto de los cambios en los objetos verificables referenciados por pruebas
El Analista necesita detectar rápidamente aquellos TCs que puedan ser inválidos, debido a cambios producidos en los objetos verificables para los cuales dichos TCs definen casos de prueba. Esta información le será suministrada bajo la forma de un análisis de impacto.

Rimp2. Reconstrucción de tablas
Al editar un TC el Analista necesita que se le avise si ya han sido impactados los cambios que pudieron haberse producido en los objetos que el TC verifica, contra dicho TC. Si los cambios aún no han sido impactados el Analista contará con la posibilidad de hacerlo. En caso afirmativo la extensión intentará la reconstrucción automatizada de todas las tablas conteniendo parámetros que sufrieron el impacto del cambio, tratando de minimizar la perdida de información. El Analista podrá intervenir para solucionar los casos que la automatización no pueda resolver.

Rimp3. Reconstrucción de *test*
Luego de un cambio en un TC deben eliminarse y generarse nuevamente, en forma automática, los objetos *testaut* que implementan la prueba.

Acerca de los algoritmos

Ralg1. Prueba de *procedures* con *ColumnTest* (CT)
Implementa un símil de *ColumnFixture* (CF) de FIT. Cada fila de la tabla contiene los parámetros para una ejecución del *procedure*. Existe una columna adicional donde el Analista puede especificar si selecciona o ignora la fila.

Elementos:

- Un *procedure* a probar.
- Tabla "TC". Sus columnas corresponden a los parámetros del procedimiento a probar más la columna para selección.

Descripción:

- Por cada fila seleccionada de la tabla: (se detiene al encontrar el fin de la tabla)
    - Se cargan los parámetros de entrada (o entrada-salida) del *procedure* según los valores contenidos en las celdas.
    - Se ejecuta el *procedure*
    - Se comparan los valores obtenidos con los resultados esperados. Si un resultado esperado ha sido dejado en blanco simplemente se mostrará el valor obtenido en el resultado sin influir en la determinación de fallo o suceso.

Ralg2. Prueba de *business components* (BC) con *ActionTest* (AT).

Implementa un símil de *ActionFixture* (AF) de FIT, donde las acciones a ejecutar corresponden a métodos que ofrece el BC a probar más comandos para la carga de valores de atributos y la comprobación de resultados.

Elementos:

- Un BC a probar.
- Tabla "AT" con 3 columnas (acción, atributo, valor esperado).

Descripción:

- Por cada fila de la tabla (se detiene al encontrar el fin de la tabla)
    - Según el valor contenido de la primera columna
        - Si es "*Enter*" cargará el valor indicado en la tercera columna en el atributo cuyo nombre se indica en la segunda columna.
        - Si pertenece a {*"Add", "Save", "Load", "Delete", "Check"*} ejecutará el método correspondiente del BC seguido de una comprobación del resultado mediante la ejecución de un método *Fail*. Si *Fail* da verdadero se devolverá como resultado el contenido del mensaje relacionado a la falla y se juzga el resultado teniendo en cuenta la segunda columna, según esta contenga o no la palabra "error" (implica en dicho caso que se esperaba un error),
        - Si pertenece a {"*Success*", "*Fail*"} ejecutará el método de igual nombre del BC. Se seguirá la semántica del caso anterior, ya que se habilitó la posibilidad de poder indicar "error" en la segunda columna.
        - Si es "*GetMessage*" obtendrá el contenido de un mensaje según el último método ejecutado, utilizando el método correspondiente del BC. La segunda columna especifica el contenido del mensaje esperado. El valor resultante se comparará con dicho valor esperado.

- Si es "*CheckVal*" obtendrá el valor del atributo especificado en la segunda columna y se comparará con el resultado esperado indicado en la tercera columna.

Ralg3 Comprobación de datos en la base de datos con *RowTest* (RT)
Implementa un análogo de *RowFixture* de FIT para comprobar datos de una tabla de la BD.

Elementos:
- Una tabla a comprobar.
- Una tabla RT: Sus columnas corresponden a los atributos de la tabla a comprobar cuyo valor se desea verificar.
- Comprobación completa: El Analista puede indicar si se debe indicar como fallo la existencia de más registros en la base de datos que los suministrados por el RT y si desea que el resultado los incluya.

Descripción:
Se obtienen las tuplas de datos desde la tabla de la BD y se comparan con las filas contenidas en la tabla RT. Las tuplas excedentes (si la comprobación es completa) se agregan a los resultados obtenidos en la presentación de los resultados de la prueba, marcándolas con la palabra "***surplus***". La ausencia de una tupla en la tabla de la BD se debe indicar también en el resultado.

Ralg4. Comprobación de datos para *Data Selectors* (DS) y *Data Providers* (DP) con *RowTest* (RT) y *ColumnTest* (CT).
Se implementa un híbrido de *RowFixture* (RF) y *ColumnFixture* (CF) de FIT.

Elementos:
- DS o DP a verificar
- Una tabla CT conteniendo una fila por cada ejecución que se desea efectuar del DS o DP correspondiendo sus columnas a los parámetros del DS o DP.
- Tantas tablas RT como filas no vacías de la tabla CT. Sus columnas corresponderán a las variables que devuelve el DS/DP.

Descripción:
- Por cada fila de la tabla CT (se detiene al encontrar el fin de la tabla)
  - Se cargan los parámetros para la ejecución desde las celdas
  - Se "ejecuta" el DS o el DP.
  - Se comprueban los datos obtenidos contra los valores expresados en la tabla RT correspondiente. Se tienen consideraciones análogas a las descritas para comprobación de tablas de la BD.

Ralg5. Inicialización de datos con SetupTest (ST).
Implementa la función básica que provee *SetupFixture* (ST) de FIT pero con un formato similar a RT. Se utilizará para inicializar tablas en la base de datos.

Elementos:
- Una tabla de la BD a inicializar.

- Una tabla RT. Sus filas corresponden a tuplas y sus columnas a los atributos.
- Indicación de vaciar la tabla de la BD previo a la ejecución (*clean*).
- Indicación de cancelar el resto de la prueba si hubo fallos en la inicialización.

Descripción:

- Se procede a vaciar la tabla, de haberse indicado tal acción.
- Por cada fila de la tabla RT (se detiene al llegar al final de la tabla o ante el primer error)
    - Se insertan tuplas en la tabla de la BD a partir de los valores en las celdas. .
    - Se comprueba para saber si fue satisfactoria o no la inserción.

Nota: Un TC puede especificar la inicialización de varias tablas mediante STs. De estar activo el control de integridad referencial las tablas se deberán cargar en el orden adecuado, de forma de no infringir dicho control. El Analista desea contar con la posibilidad de levantar dicho control para la ejecución de una prueba dada.

Acerca de los editores de tablas

Redi1. Editores de tablas
El Analista editará las tablas con los parámetros para las pruebas. La edición debe presentar una grilla adecuada a la *fixture* que implementará la prueba deducida automáticamente por la herramienta. Se deberá establecer una convención de formatos para el ingreso (por ejemplo para las fechas y los SDT). El Analista no podrá modificar el formato de las tablas de parámetros.

Redi2. Editor de tablas para CT
El Analista necesitará editar tablas de tipo CT. El formato de la tabla sigue los lineamientos de *ColumnFixture* de FIT.

- Las columnas de dicha tabla se corresponderán, en orden, a cada uno de los parámetros del objeto a probar.
- Para cada columna se debe informar al Analista acerca del tipo de datos, dimensión, si es de entrada, entrada-salida o de salida, y el nombre del parámetro (variable o atributo) que representa.
- El ingreso de datos esperados es opcional (de no ser ingresados simplemente la prueba devolverá el resultado obtenido no influyendo en la determinación del resultado general de la prueba).

El Analista necesitará visualizar rápidamente la regla *parm* del objeto a probar durante la edición de la tabla de parámetros.

Redi3. Editor de tablas para RT
El Analista necesitará editar tablas de tipo RT. El formato de la tabla sigue los lineamientos de *RowFixture* de FIT.

- Las columnas de dicha tabla se corresponderán, en orden, a cada uno de los atributos o variables a verificar en una selección de datos (DP, DS) o tabla de la base de datos.
- De cada columna el Analista podrá saber el tipo de datos, dimensión, y el nombre del atributo correspondiente.

- Ingresa en cada celda un valor esperado.
- Puede especificar que le importa el orden de las filas para la comprobación.

El Analista necesitará tener visible rápidamente, mientras edita la tabla de parámetros, la estructura de la transacción a partir de la cual se define la tabla de la BD a comprobar.

Redi4. Editor de tablas para AT
El Analista necesitará de un editor para ingresar datos en la tabla de un AT. El formato de la tabla sigue los lineamientos de *ActionFixture* de FIT.

- Es una tabla de tres columnas.
- En la primer columna el Analista podrá indicar comandos, seleccionable desde una lista: *Enter, Add, Save, Load, Delete, Fail, Success, Check, CheckVal* y *GetMessage*. El editor validará en lo posible la validez del orden en que se expresan los comandos.
- En la segunda columna se permitirá que se elija un atributo del BC o la palabra “error”. Esto será posible para las acciones *CheckVal* y *Enter*.
- En la tercera columna se ingresa un valor esperado, para las acciones *Fail* y *GetMessage*.

El Analista necesitará visualizar la estructura de la transacción asociada al BC, durante la edición de la tabla que contiene los parámetros.

Redi5. Ingreso de datos estructurados (SDT)
El Analista necesitará que los editores de ingreso de datos permitan el ingreso de datos estructurados (SDT) para parámetros de entrada y resultados esperados. Adicionalmente, mientras edita, el Analista necesitará la posibilidad de visualizar rápidamente las estructuras de los SDTs involucrados.

Redi6. Visualización de datos estructurados (SDT)
El Analista necesitará visualizar los resultados detallados de las pruebas en las que intervienen SDT, expandiendo los resultados esperados y los obtenidos, de forma de lograr su representación conjunta y fácilmente descubrir donde no hubo coincidencias. Toda información de resultado que excede lo esperado debe identificarse como “***surplus***”, salvo se indique lo contrario por parte del Analista.

Redi7. Facilidades de los editores
El Analista ingresará y editará los datos contenidos en las tablas de parámetros bajo control apropiado de tipos y sugerencias de valores preestablecidos. Necesita insertar nuevas filas, eliminar filas, copiar o cortar y pegar filas enteras, repetir filas con valores anteriores. El Analista podrá moverse entre celdas utilizando el tabulador y las teclas de dirección y ordenar las filas según valores de sus columnas. Podrá también eliminar una tabla entera de parámetros.

Acerca de Listas y referencias cruzadas

Rlyr1. Listas y referencias cruzadas de TCs
El Analista necesita identificar y ubicar rápidamente los TCs y dentro de ellos diferenciar eficazmente sus diferentes componentes.

Rlyr2. Listas y referencias cruzadas de objetos bajo prueba
El Analista necesita identificar y ubicar rápidamente los objetos verificables invocados por los TCs. Dado un objeto cualquiera querrá poder identificar y ubicar rápidamente los TCs que lo referencian.

Acerca de la Exportación/Importación

Reyi1. Exportación de TCs
Se podrán exportar TCs. Se exportarán junto con los valores de sus parámetros.

Reyil1. Importación de TCs, *fixtures* y tablas de parámetros
Se podrá importar TCs. Se importarán junto con los valores de sus parámetros. El análisis de la importación determinará incongruencias y podrá abortar la importación.

Acerca de Build

Rbui1. *Builds* de TCs
El Analista hará el *build* de todos o de algunos de los objetos que implementan pruebas. Al especificar un TC se valida tal como se realiza con cualquier otra instancia de objeto GeneXus sometida a especificación, desde el punto de vista de su sintaxis, coherencia y en este caso según las *fixtures* que se implementarán. El resultado final incluirá la especificación de todos los objetos *test* involucrados o derivados del TC. La generación de un TC disparará la generación de las instancias de los objetos *test* necesarios para la prueba.

Rbui2. Exclusión de los objetos para pruebas
El Analista debe contar con un mecanismo simple que le permita ignorar todos los objetos para prueba en un *build,* especialmente para cuando genera una implementación del sistema.

Acerca de los dobles para pruebas

Los siguientes relatos enunciarán la primera aproximación a la solución del problema sobre el aislamiento de las pruebas, utilizando la técnica de sustitución. Se seguirá la línea precedente de adaptación de funcionalidades de FIT.

Rmoc1. *Mock*
El Analista podrá indicar para un objeto GeneXus un comportamiento sustitutivo del nativo, definido por código (*source*) y una tabla de parámetros que se denominará "tabla de expectativas". Esta información se conocerá como **"*mock*"** del objeto.

Ejecutar el objeto bajo dicho comportamiento alternativo se conocerá como ejecución en "modo *mock*".

Rmoc2. Detección de colaboradores
El Analista necesitará obtener una lista donde se identifiquen todos los objetos invocados directamente o indirectamente por el objeto a probar. De esta lista de objetos invocados necesitará rápidamente reconocer cuales no tienen asociada una función sustitutiva.

Rmoc3. Determinación de *mocks* para un TC
El Analista podrá indicar para un TC de un *procedure* cuales objetos GeneXus de los invocados ejecutarán en modo *mock* durante la próxima ejecución de su prueba.

Rmoc4. Especificación de expectativas
La tabla de expectativas es de formato similar a una tabla CT. Contiene una columna por cada parámetro de entrada y cada parámetro de salida, y dos columnas por cada parámetro de entrada-salida (*in:out*) para representar ambos valores en forma independiente, además de las siguientes columnas adicionales:

- Una columna donde marcar la selección de la fila para la próxima prueba.
- Una columna con un valor entero que se decrementa en 1 por cada invocación, inicializado en 9s, rotulada como "devoluciones".
- Una columna, rotulada "invocación".

Una de las filas de la tabla podrá ser identificada por el Analista como contenedora de "valores de respuesta por defecto". En dicha fila los valores de los parámetros de entrada no se especifican.

Rmoc5. Exhibición de resultados
Los resultados de cada ejecución en modo *mock* de los objetos invocados durante la prueba de un *procedure* se presentarán, en el informe de la ejecución del TC, junto a los resultados de cada ejecución de dicho *procedure*. El formato del resultado respetará en lo posible la misma representación tabular de las expectativas, utilizando la semántica de colores habitual.

Rmoc6. Comportamiento de un *mock*
Cada vez que el objeto es invocado bajo un comportamiento *mock* ejecutará el siguiente algoritmo, adaptado de CT:

<u>Elementos:</u>
La tabla de expectativas.
Código alternativo (*source* del *mock*)
<u>Descripción:</u>

- Por cada fila de la tabla CT (excluyendo las filas no seleccionadas):
  - Incrementa un contador de invocaciones.
  - Confronta los valores de los parámetros de entrada con los valores que los representan en la fila de la tabla CT (considera para ello los

operadores que se indicaron en las celdas, por defecto usa la igualdad).

- Si cumplen la condición:
    - Carga los parámetros de salida con los valores de los resultados esperados en la tabla CT para esa fila.
    - Decrementa el contador de devoluciones de esa fila.
    - Si el contador de invocaciones no coincide con el valor especificado en la columna "invocación" indicará un "fallo", pintando de rojo en el informe de resultados la columna "invocación". Si coincide la pinta de verde.
    - Pinta de verde el resto de las columnas de la fila en el informe de resultados y finaliza la recorrida de la tabla CT.
- Si no cumple la condición itera (otra fila de la tabla)

- Si al finalizar la ejecución los parámetros de salida no han sido aún cargados los carga con el contenido de la fila que almacena los "valores por defecto" y pinta de verde, en el informe de resultados, a dicha fila. Si el contador de invocaciones no coincide con el especificado para dicha fila lo marca en rojo en el informe de resultados.
- Marca en el informe de resultados aquellas celdas de la columna del contador de devoluciones que quedaron negativas luego de la ejecución.
- Ejecuta el código suministrado por el Analista en *source*.

Rmoc7. Indicación de éxito o fracaso.
El marco de pruebas deberá ser notificado, luego de la ejecución en modo *mock* de un objeto, acerca del resultado global de su ejecución. Si alguna celda resultó coloreada en rojo se considera la ejecución como fallida. De la misma forma el código suministrado por el Analista en el *source* podrá indicar que el resultado es fallido. Se considerará fallida la prueba del TC si la ejecución en modo *mock* de alguno de los objetos invocados indicó una falla.

Rmoc8. Vuelco de información en la bitácora
El código suministrado por el Analista en el *source* podrá almacenar información en el informe de resultados.

Rmoc9. Impacto del cambio
Los cambios en la regla *parm* o en sus parámetros impactarán en el *mock.* Se deberán reorganizar de forma automatizada las tablas con expectativas para recoger los cambios.

Se considera que el conjunto de relatos precedentes es suficiente para la elaboración de una versión inicial de la herramienta de pruebas unitarias con valor funcional para los Analistas GeneXus, incorporando funcionalidades inspiradas en las *fixtures* básicas de FIT.

7.5 Modelo de Dominio

En esta sección se describirá un modelo de dominio para la herramienta especificada en la presente propuesta y se establecerá una trazabilidad hacia los relatos que la especifican (7.4).

Un modelo de dominio es utilizado para modelar el dominio de un problema cuando se desarrolla un sistema computacional [SØ05]. En relación a la propuesta, el dominio es la prueba unitaria de objetos GeneXus sin interfase de usuario y los requerimientos para el sistema los expresados bajo la forma de relatos (*stories*) en (7.4). En base a ellos se elabora el modelo de alto nivel que se expone en la figura 7-7-2, el cual exhibe los conceptos principales y sus relaciones.

En el modelo se muestra el concepto Caso de Prueba (TC), el cual está directamente vinculado con los relatos royp2 y royp7 a royp10. En dichos TCs el Analista especificará lo necesario para probar objetos GeneXus verificables, los que se visualizan como los conceptos BC, Procedimiento (*Procedure*) sin UI, DS y DP según los relatos royp1 y rfix1.

Según se deba verificar datos o según sea la clase de objeto a probar, se aplicarán diferentes algoritmos correspondientes a la verificación de datos o a la clase de objeto a probar. Se identifican en el modelo con los conceptos AT, CT, RT y ST según los relatos rfix1, rfix2, rfix3, rfix4, ralg1 a ralg5. Estas pruebas se implementan con los objetos GeneXus generados y mantenidos en forma automatizada a partir del Caso de Prueba (TC) según royp3, rimp1 y rimp3.

El Analista puede suministrar procedimientos adicionales para realizar inicialización de datos, verificación de datos y reestablecer el ambiente al final de la ejecución, lo cual se expresa en los conceptos PVU, PIU y PFU según los relatos royp13 a royp17.

De la ejecución de las pruebas implementadas a partir de un TC se obtienen resultados y un reporte de la ejecución, representados en el modelo por los respectivos conceptos, siguiendo los relatos rius1 a rius4. Opcionalmente dichos resultados y reportes de ejecución pueden ser almacenados en una bitácora, según los relatos rbit1 a rbit4, que queda accesible al TC para su visualización desde el IDE.

Para las pruebas se suministran parámetros expresados en tablas, que se crean y editan desde los TC, conteniendo los datos de entrada, comandos y resultados esperados para las pruebas, para las cuales la herramienta asegura el formato adecuado e intenta preservar los datos ante cambios que obliguen a su reconstrucción, según los relatos rfix4, rfix5, redi1 a redi7 y rimp1 a rimp3.

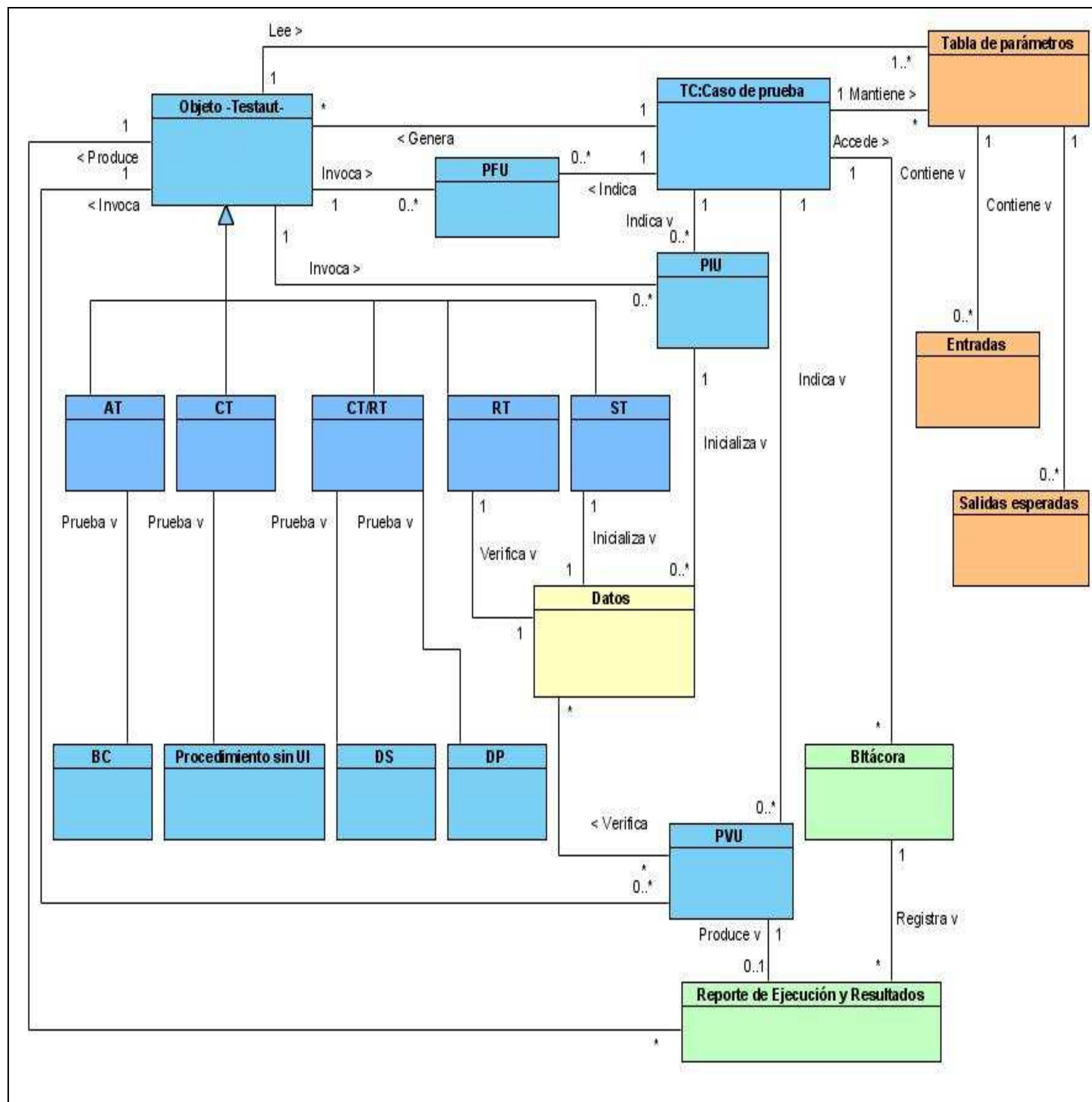


Fig. 7- 2: Modelo de dominio general de GXUnit-GXFIT.

El Analista puede crear y codificar objetos verificadores de "usuario" e identificarlos con una propiedad *test*. Los objetos para verificación que produzca automáticamente la herramienta para implementar *fixtures* pertenecerán a una sub categoría *testaut.* En el diagrama de la figura 7-3 se ilustra acerca de la relación entre los conceptos referidos a objetos verificadores *test, testaut* y *fixtures*, según relatos royp2, royp3, royp 13 a royp15 y rfix1.

En cuanto a las *fixtures***,** definidas según relatos rfix1 a rfix4, estas necesitan de diferentes formatos de tablas de parámetros, tal como narra el relato rfix5. En dichas tablas el Analista especificará los parámetros para conducir las pruebas. El diagrama de la figura 7-4 ilustra al respecto de la relación entre las tablas de parámetros, los formatos de tablas y las *fixtures* correspondientes.

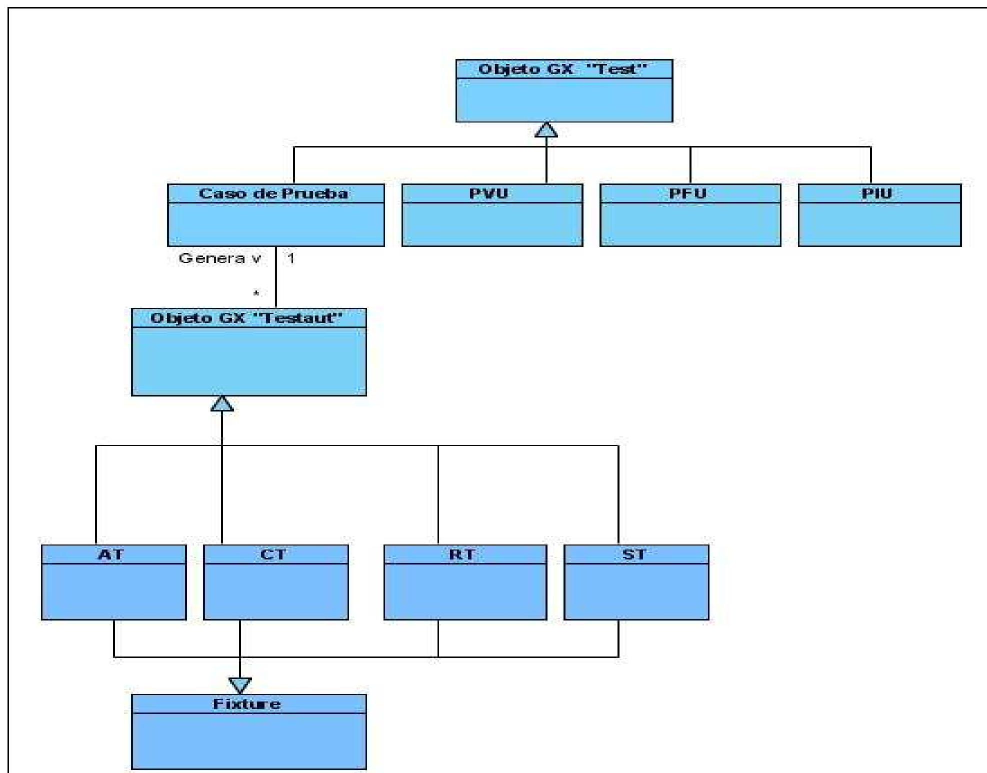


Fig. 7- 3: Modelo de objetos verificadores GXUnit-GXFIT.

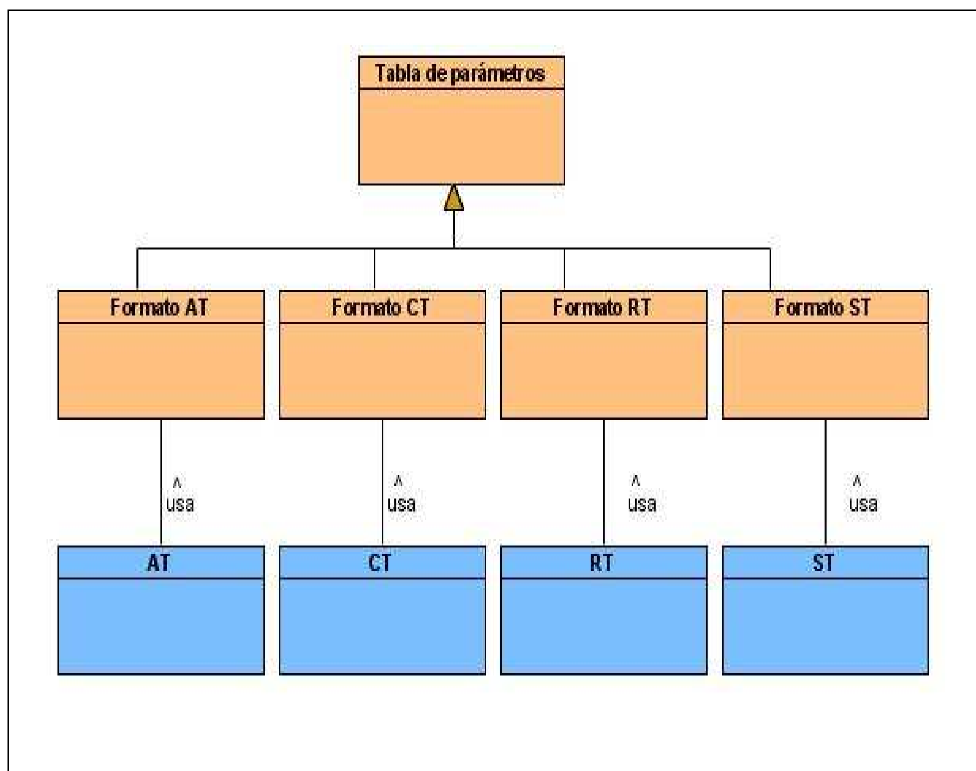


Fig. 7- 4: Modelo de tablas de parámetros GXUnit-GXFIT.

Un caso de prueba está compuesto de diferentes partes, según relatos rfix4 y rmoc3. El Analista utiliza dichas partes para especificar los diferentes elementos (varios de ellos son opcionales) componentes de la prueba unitaria. El diagrama de la figura 7-5 ilustra los conceptos relacionados a este respecto.

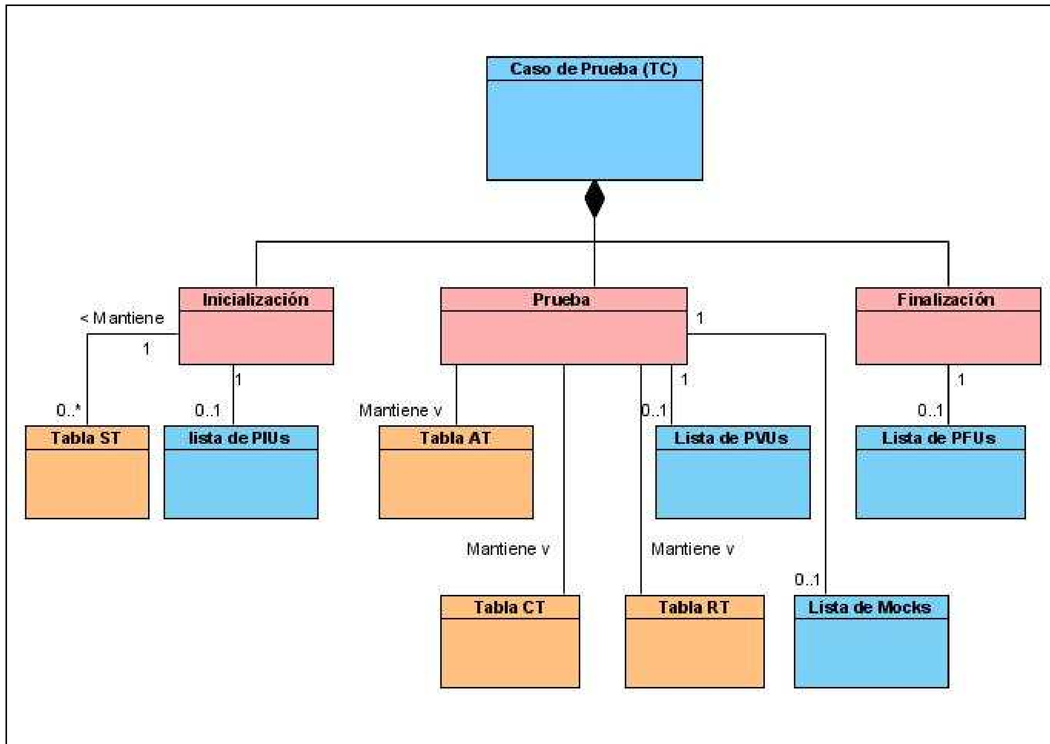


Fig. 7- 5: Modelo de un caso de prueba GXUnit-GXFIT.

En relación a los dobles para pruebas, la figura 7-6 les representa en un modelo, según relatos rmoc1 y rmoc4.

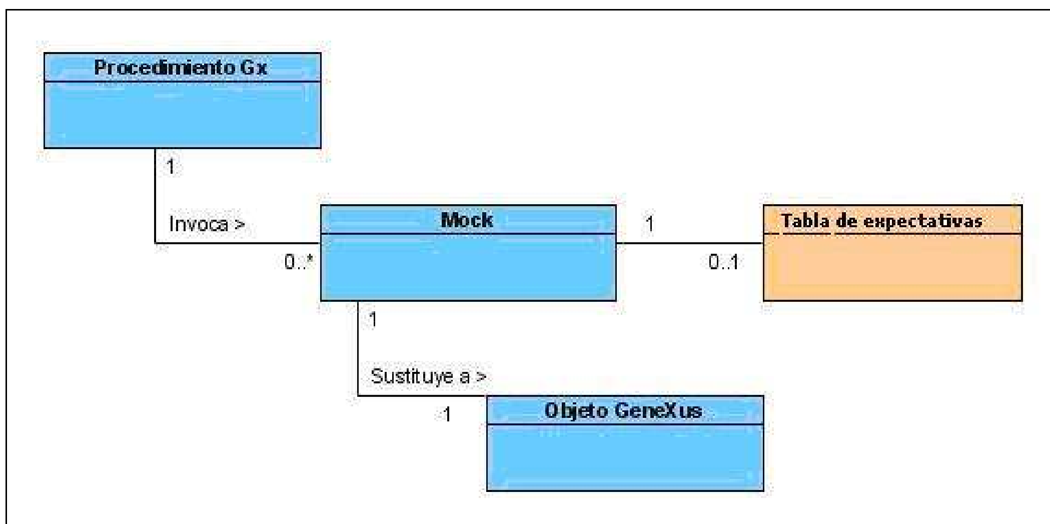


Fig. 7- 6: Modelo de dobles para pruebas GXUNIT-GXFIT.

C a p í t u l o 8

CONCLUSIONES

Visto que no existía una herramienta de pruebas unitarias específica para ambientes de desarrollo GeneXus se abordó el tema. El objetivo central consistió en brindar un documento de especificación para una herramienta de tales características. La especificación realizada recogió funcionalidades inspiradas en el *FrameWork for Integrated Tests* (FIT).

El trabajo realizado puede resumirse en tres resultados: la obtención de un resumen del estado del arte sobre las pruebas, la obtención de un documento de especificación de una herramienta de pruebas inspirada en FIT para su utilización especifica en el desarrollo con GeneXus y la participación (compartida) en la construcción de los primeros prototipos de la herramienta.

En este capítulo se describen los objetivos alcanzados, proponiéndose investigación y trabajo a futuro. Se organiza de la siguiente manera:

- En la sección 8.1 se expone acerca de los objetivos alcanzados.
- En la sección 8.2 se enumeran aportes y limitaciones.
- En la sección 8.3 se propone investigación y trabajo a futuro.

8.1 Objetivos alcanzados

Visto que no existía una herramienta de pruebas unitarias específica para utilizar bajo el desarrollo en GeneXus se decidió abordar el tema con el objetivo de aportar un documento de especificación para la misma.

La especificación realizada recogió funcionalidades inspiradas en el *FrameWork for Integrated Tests* (FIT) y las adaptó a la definición de pruebas unitarias en GeneXus, en el entendido que dichas funcionalidades constituyen una buena técnica para la definición de pruebas unitarias y que se viabiliza su utilización si se aplican las facilidades de reconstrucción y generación de código habituales en GeneXus.

Se elaboró dicha especificación a partir de una investigación guiada por las siguientes preguntas: ¿es posible crear un marco para automatizar las pruebas unitarias de objetos GeneXus? ¿qué requerimientos debería cumplir dicho marco para aprovechar las características de GeneXus en la elaboración de programas para prueba y el mantenimiento de los casos de prueba en forma automatizada? ¿cómo adaptar funcionalidades inspiradas en FIT para dicho marco de pruebas unitarias de programas producidos por GeneXus teniendo en cuenta sus particularidades, posibilidades y características?

En respuesta a dichas interrogantes se obtuvo el documento de especificación de requerimientos y modelo de dominio contenido en este trabajo, dando cumplimiento a los objetivos específicos detallados en el alcance del proyecto, y se participó activamente en el rol (compartido) de Cliente de dos equipos de desarrollo encargados de la construcción de los primeros prototipos de la herramienta.

8.2 Aportes y limitaciones

El primer aporte lo constituyó la obtención de un resumen del estado del arte relacionado a las pruebas de *software*. Se presentaron las definiciones y objetivos de las pruebas. Se describieron las consideraciones a ser tenidas en cuenta al probar y los diferentes tipos de pruebas. Se introdujo a la verificación y validación bajo la óptica de las metodologías ágiles, en el entendido que son de aplicación sus principios al desarrollo con GeneXus, resumiéndose el enfoque ante las pruebas de cuatro de ellas: XP, SCRUM, FDD Y DSDM. Se estudió acerca de *Test Driven Development*. Se introdujo a la automatización de las pruebas en general haciendo énfasis en la prueba unitaria y funcional utilizando XUnit y FTT respectivamente, dada su relación con la propuesta contenida en este trabajo. Se resumió finalmente sobre FIT y GeneXus.

El segundo aporte se inscribió en la participación en el rol compartido de Cliente, con la responsabilidad de especificar la primera versión de una herramienta de pruebas unitarias, denominada GXUnit, para utilizar bajo GeneXus, efectuando el seguimiento de la construcción de los prototipos y su posterior aprobación.

La tercera contribución se constituyó con la obtención del documento de requerimientos y modelo de dominio para la especificación de un conjunto de funcionalidades comunes con la herramienta GXUnit y otro conjunto específico (GXFIT) de funcionalidades inspiradas en FIT, como complemento de GXUnit.

Los 65 relatos que se brindaron en el documento de especificación se concentraron en el abordaje de los aspectos que se consideraron esenciales. El documento obtenido se limitó a la consideración de las principales características que podría poseer una herramienta de pruebas unitarias para ambientes de desarrollo GeneXus, de valor para los Analistas GeneXus, adaptando funcionalidades de FIT. Su alcance se restringió a la prueba de objetos sin interfase de usuario.

8.3 Investigación y trabajo futuro

Esta sección presenta el trabajo que se puede realizar a futuro. La forma de continuar puede tomar varios caminos, dentro de las vertientes enmarcadas por la evolución de la herramienta, su aplicación práctica y el alcance de la especificación.

En cuanto a la evolución de la herramienta el primer camino está dado por el desarrollo de los prototipos de GXUnit por equipos externos a Artech, en forma incremental hasta transformarlos en herramientas que otorguen valor para los Analistas GeneXus. Es en este sentido que se considera muy oportuno que nuevos grupos de estudiantes, como parte de su actividad académica, pudieran tomar a su cargo la construcción de estos nuevos prototipos. El segundo camino está determinado por el desarrollo de funcionalidad para pruebas unitarias que la propia empresa Artech estimara pertinente realizar en GeneXus.

En cuanto a la aplicación de la herramienta se entiende necesario realizar varios estudios de campo, una vez que alcance un nivel de madurez que los permita.

En cuanto a la especificación, se deberá ampliar el alcance con el objetivo de abarcar los objetos GeneXus aún no contemplados, especialmente aquellos con interfase *web* y *win*, en ese orden.

El código fuente de los primeros prototipos de GXUnit elaborado por los dos grupos de estudiantes ha sido liberado al dominio público para que otros grupos de desarrolladores puedan conocerlo y efectuar sus aportes. El trabajo más inmediato deberá consistir en hacerlos operativos para GeneXus X; la unificación de los dos prototipos actuales se entiende que también debería ser considerada.

Por último, el documento de especificación de GXUnit/GXFIT contenido en este trabajo de tesis será publicado, estimándose que pueda convertirse en un importante insumo para el proyecto.

Bibliografía

[Aae07] Aaen I., “Up, Patterns & FitNesse”, Departament of Computer Science, Aalborg University. Denmark, http://www.cs.aau.dk/~ivan/SOE2007/MM6.pdf, 2007. Último acceso: 10/02/2008.

[Abb06] Abbot Java Gui Test Framework, “Getting Started with the Abbot Java GUI Test Framework”, http://abbot.sourceforge.net/doc/overview.shtml, 2006. Último acceso: 19/01/2008.

[Abr02] Abrahamsson P., Salo O., Ronkainen J., Warsta J., “Agile software development methods:Review and Analysis” Espoo, Finland: Technical Research Centre of Finland, VTT Publications 478, http://oasis.oulu.fi/publications/vtt-pa.pdf, 2002. Último acceso: 10/02/2008.

[Abr03] Abrahamsson P.,Warstab J.,Siponenb M.,Ronkainena J. “New Directions on Agile Methods: A Comparative Analysis”**,** Proceedings of the International Conference on Software Engineering, Portland, Oregon, USA, May 3-5, 2003. http://agile.vtt.fi/docs/publications/2003/2003_icse03_new_directions_on_agile_methods.pdf, Último acceso: 10/02/2008.

[Ada07] Adams T. “Better Testing Through Behaviour”, Open Source Developer’s Conference, Nov. 2007, Brisbane, Australia, http://adams.id.au/blog/wp-content/uploads/2007/10/OSDC2007BetterTestingThroughBehaviour.pdf. Último acceso: 31/01/2008.

[ADM96] Advanced Development Methods Inc. “Controlled Chaos: Living in the Edge”, OOPSLA’96, San Jose, California, USA, http://jeffsutherland.com/oopsla96/schwaber.html, 1996. Último acceso: 31/01/2008.

[Adz107] Adzic G. “Getting Fit with Net. Quick Introduction to Testing .Net Applications with FitNesse”, Version 0.2, http://www.gojko.net/FitNesse/fitnesse.pdf, 2007. Último acceso: 10/02/2008.

[Adz207] Adzic G. “Getting FIT with Databases”, http://dbfit.svn.sourceforge.net/svnroot/dbfit/dbfit/docs/dbfit.pdf, 2007. Último acceso: 10/02/2008.

[Adz307] Adzic G. “Automated web test with FitNesse and Selenium”, http://gojko.net/2007/05/20/automating-web-tests-with-fitnesse-and-selenium, 2007. Último acceso: 10/02/2008.

[Agu05] Aguiar A. “Right here, right now” “Test Driven Development vs working at the right level of abstraction”, http://weblogs.asp.net/aaguiar/archive/2005/12/05/Test-Driven-Development-vs-working-at-the-right-level-of-abstraction.aspx, 2005. Último acceso: 10/02/2008.

[Alm03] Almeida E. "Desarrollando desde la Trinchera", "Desarrollo con GeneXus: En el proceso de crear un *framework* de *testing*", http://ealmeida.blogspot.com/2003/10/estoy-en-el-proceso-de-crear-un.html, 2003. Último acceso: 10/02/2008.

[Alm04] Almeida E. "Software Testing: Tres enfoques para un mismo problema", Conferencia en XIV Encuentro Internacional de Usuarios GeneXus, http://www.concepto.com.uy/archivosvinculados/EncuentroGX2004SoftwareTesting.ppt, 2004. Último acceso: 10/02/2008.

[And07] Andrea J. "Envisioning the Next Generation of Functional Testing Tools", IEEE Computer, Vol.24, Issue 3, pp. 58-66, May/Jun 2007.

[Ant07] Apache Software Fundation "The Apache Ant Project", "Welcome", http://ant.apache.org/, 2007. Último acceso: 10/02/2008.

[Ame07] Endesa "Ejemplo Eclipse – JUnit 4", http://ame.endesa.es/confluence/display/AMEDev/Ejemplo+Eclipse+-+Junit+4, 2007. Último acceso: 10/02/2008.

[Ara07] Araújo A. "Test Driven Development. Fortalezas y Debilidades", Serie reportes técnicos, ISSN: 0797–6410, Instituto de Computación, Facultad de Ingeniería, Pedeciba Área Informática, Universidad de la República, Uruguay, http://www.fing.edu.uy/inco/pedeciba/bibliote/reptec/TR0713.pdf, 2007. Último acceso: 10/12/2007.

[Art105] Artech Consultores S.R.L. "Visión general de GeneXus", http://www.GeneXus.com/portal/agxppdwn.aspx?2,32,660,O,S,0,22790%3bS%3b1%3b2315, 2005. Último acceso: 10/02/2008.

[Art205] Artech Consultores S.R.L. "Primeros pasos con GeneXus 9.0", http://www.genexus.com/portal/agxppdwn.aspx?2,32,660,O,S,0,22792%3bS%3b1%3b2315, 2007. Último acceso: 10/02/2008.

[Ast03] Astels D. "Test-Driven Development: A Practical Guide", ISBN 0131016490, Prentice Hall, 2003.

[Ast05] Astels D. "A New Look at Test-Driven Development", http://blog.daveastels.com/files/BDD_Intro.pdf, 2005. Último acceso: 31/01/2008.

[Bac00] Bach, J. "Session-Based Test Management", http://www.satisfice.com/sbtm, Ültimo acceso: 20/02/2008.

[Bac03] Bach J. "Exploratory Testing Explained", http://www.satisfice.com/articles/et-article.pdf, 2003. Ültimo acceso: 20/02/2008.

[Bac97] Bach J. "Good Enough Quality: Beyond the Buzzword," IEEE *Computer*, Vol. 30, Issue 8, pp. 96-98, Agosto 1997.

[Bac98] Bach J. "A Framework for Good Enough Testing", IEEE Computer, Vol.31, Issue 10, pp.124-126, Octubre 1998.

[Bac99] Bach J. "Risk-based Testing", Software Testing and Quality Engineering Magazine Vol. 1, No. 6, November-December 1999, www.stickyminds.com/getfile.asp?ot=XML&id=5009&fn=Smzr1XDD1800filelistfilename1.pdf, Último acceso: 31/01/2008.

[BA04] Beck K. , Andres C. ""Extreme Programming Explained, Embrace Change 2nd Edition", ISBN 0-321-27865-8, Addison Wesley, 2004.

[Bec02] Beck K. "Test Driven Development by Example", ISBN 032-114653-0, Addison Wesley, 2002.

[Bec03] Beck K. "Scale-Free Extreme Programming", http://www.cse.yorku.ca/course_archive/2003-04/W/6442/misc/Kent Beck scale free.pdf , 2003. Último acceso: 10/02/2008.

[Bec94] Beck K. "Simple Smalltalk Testing: With Patterns", First Class Software, Inc., http://www.xprogramming.com/testfram.htm, 1994. Último acceso: 10/02/2008.

[Bec199] Beck K. "Extreme Programming Explained, Embrace Change",ISBN 201-61641-6, Addison Wesley, 1999.

[Bec299] Beck K. *"Embracing Change with Extreme Programming"*, IEEE Computer, pp. 70-77, Octubre 1999.

[Bee97] Beedle M. "SCRUM is an Organization Pattern", http://jeffsutherland.com/objwld98/ow_scrum.html, 1997. Último acceso: 10/02/2008.

[Bee98] Beedle M., Devos M., Sharon Y., Schwaber K., Sutherland J. "SCRUM: An extension pattern language for hyperproductive software development", 1998.

[Bei90] Beizer B. "Software testing techniques", 2nd. Edition, ISBN 0-442-20672-0, Van Nostrand Reinhold Co, 1990.

[Bel06] Bellware S. "DSL's for Testing; Behavior-Driven Design; FitNesse - A Conversation with Bret Pettichord", http://codebetter.com/blogs/scott.bellware/archive/2006/09/09/149105.aspx, 2006. Último acceso: 02/11/2007.

[Bel07] Bellware S. "The Problem with Writing Acceptance Tests in a Custom DSL", http://codebetter.com/blogs/scott.bellware/archive/2007/10/28/170346.aspx, 2007. Último acceso: 02/11/2007.

[Ber03] Bergin J. "XP Testing a GUI with FIT, Fitnesse, and Abbot", Pace University, http://www.csis.pace.edu/~bergin/xp/guitesting.html, 2003. Último acceso: 19/1/2008.

[BF00] Beck K., Fowler M. “Planning Extreme Programming”, ISBN 0-201-71091-9, Addison Wesley, 2000.

[BG99] Beck K., Gamma E.. “JUnit A Cook's Tour”, http://junit.sourceforge.net/doc/cookstour/cookstour.htm, 1999. Último acceso: 20/02/2008.

[Bin99] Binder R. “Testing Object-Oriented Systems: Models, Patterns, and Tools”, ISBN 0-201-80938-9, Addison-Wesley Professional, 1999.

[Boe84] Boehm B. "Verifying and Validating Software Requirements and Design Specifications", IEEE Software, Volume 1, pp. 75-88, 1984.

[Bro95] Brooks F. P. “The Mythical Man-Month: Essays on Software Engineering”, Anniversary Edition, ISBN 0-201-83595-9, Addison Wesley, 1995.

[Bur03] Burnstein I. “Practical Software Testing. A Process-Oriented Approach”, ISBN 0-387-95131-8, Springer-Verlag New York, Inc., 2003.

[BV02] Belzarena P. , Von Sanden R. “Una propuesta metodológica para alinear los proyectos de la empresa con su estrategia”, Anales del Congreso Latinoamericano de Estrategia. Montevideo, Uruguay, 2002.

[BT03] Boehm B., Turner R. “Balancing Agility and Discipline: A Guide for the Perplexed”, ISBN 0-321-18612-5, Addison Wesley, 2003.

[C304] C3 Project “Chrysler Comprehensive Compensation” http://c2.com/cgi/wiki?ChryslerComprehensiveCompensation, 2004. Último acceso: 10/02/2008.

[C3T98] The "C3 Team". “Chrysler Goes to ‘Extremes’ ”, Distributed Computing, Octubre 1998, pp. 24 - 26, http://www.xprogramming.com/publications/dc9810cs.pdf, Último acceso: 10/02/2008.

[Cas07] Castagnet N. **“**Software Factories para construir Sistemas de Información con GeneXus”, Informe del proyecto de grado, Tutor: Triñanes, J. ,Instituto de Computación, Facultad de Ingeniería, Universidad de la República. http://www.fing.edu.uy/inco/grupos/gris/wiki/uploads/Ensenianza/InformeFinal-Nc2006.pdf, 2007. Último acceso: 1/12/2007.

[CDT05] Context-Driven Testing School http://www.context-driven-testing.com, 2005. Último acceso: 10/02/2008.

[CDT07] Context-Driven Testing “What is Context-Driven Testing?”, http://c2.com/cgi/wiki?ContextDrivenTesting , 2007. Último acceso: 1/12/2007.

[CH02] Crispin L., House T. “Testing Extreme Programming”, ISBN 0-321-11355-1, Addison Wesley, 2002.

[CMMI02] "Capability Maturity Model Integration (CMMI)", Version 1.1, CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1), Continuous Representation CMU/SEI-2002-TR-011 ESC-TR-2002-011, 2002.

[Cli04] Clifton M. "Advanced Unit Test, Part V – Unit Test Patterns" "An ntroduction To The Concept of Unit Test Pattern", http://www.codeproject.com/gen/design/autp5.asp, 2004. Último acceso: 1/12/2007.

[Cob07] Jcoverage ltd., "Cobertura", http://cobertura.sourceforge.net/, 2006. Último acceso: 10/02/2008.

[Coc00] Cockburn A. "Reexamining the Cost of Change Curve, year 2000",http://www.xprogramming.com/xpmag/cost_of_change.htm, 2000. Último acceso: 10/02/2008.

[Cod70] Codd E. "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, Vol.13, No.6, Junio 1970, pp 377-387, Association for Computing Machinery, Inc.

[Con06] BandXI International "conFIT" "Fitnesse for Eclipse Plugin", http://www.bandxi.com/fitnesse/index.html, 2006. Último acceso: 10/02/2008.

[Cri07] Crispino M., "Demos sobre *GxExtensions*", Demostración acerca de la construcción de GxExtensions realizada durante la tercera reunión de usuarios GeneXus de Montevideo, Uruguay, 26/05/2007. http://video.google.es/videoplay?docid=6608025377315378664. Último acceso: 10/02/2008.

[Csh06] ISO-IEC International Organization for Standarization- International Electrotechnical Commission, ISO-IEC 23270 2nd. Edition, http://standards.iso.org/ittf/PubliclyAvailableStandards/c042926_ISO_IEC_23270_2006(E).zip, 2006. Último acceso: 10/02/2008.

[Cun107] Cunningham W. "Tips for Core Implementors" "Framework for Integrated Tests",http://fit.c2.com/wiki.cgi?TipsForCoreImplementors , 2007. Último acceso: 10/02/2008.

[Cun207] Cunningham W. "Welcome Visitors", "Framework for Integrated Tests", http://fit.c2.com/, 2007. Último acceso: 10/02/2008.

[Cun307] Cunningham W. "FrameWork History", http://fit.c2.com/wiki.cgi?FrameworkHistory, 2003. Último acceso: 10/02/2008.

[CW01] Cockburn A., Williams L. "The Cost and Benefits of Pair Programming", http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF, 2001 Último acceso: 28/09/2007.

[DA02] Díaz Arnesto P., Araújo A. *"*Propuesta para la Mejora del Proceso de Desarrollo de Software en Pequeñas Empresas*"*, Trabajo Final del Curso Ingeniería de Software, Docente: Brum F., CPAP, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2002.

[Dij70] Dijkstra E. "Notes on structures Programming", TH Report 70 –WSK-03, http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF, 1970. Último acceso: 1/12/2007.

[Dos05] Doshi G. "JUnit in 10 minutes: a Quick Reference Guide", http://www.instrumentalservices.com/media/articles/java/junit4/JUnit4.pdf, 2005. Último acceso: 10/02/2008.

[DSDM04] DSDM Consortium "Practical testing guidelines", http://www.dsdm.org/knowledgebase/download/53/practical_testing_guidelines.pdf, 2004. Último acceso: 10/02/2008.

[DSDM107]
DSDM Consortium " DSDM CONSORTIUM", http://www.dsdm.org, 2007. Último acceso: 10/02/2008.

[DSDM207]
DSDM Consortium "Testing", http://www.dsdm.org/version4/2/public/Testing.asp, 2007. Último acceso: 10/02/2008.

[Dus03] Dustin E., "Effective Software Testing. 50 Specific Ways to Improve your Testing", ISBN 0-201-79429-2, Addison Wesley, 2003.

[DXP02] Kircher M. et al. "Distributed eXtreme Programming", http://www.kircher-schwanninger.de/michael/publications/xp2001.pdf, 2001. Último acceso: 10/02/2008.

[Eas08] EasyMock , http://www.easymock.org/, 2008. Último acceso: 31/01/2008.

[Ecl07] Eclipse Foundation "Eclipse - an open development platform" http://www.eclipse.org, 2007. Último acceso: 10/02/2008.

[Eva04] Evan E. "Domain-Driven Design: Tackling Complexity in the Heart of Software", ISBN 0-321-12521-5, Addison Wesley, 2004.

[FDA-GSV-V2]
Food and Drug Administration "General Principles of Software Validation; Final Guidance for Industry and FDA Staff- V2", 2002.

[FDD03] Nebulon Pty Ltd. "Feature Driven Development Forums. Test Driven Development versus Defensive Coding", http://www.featuredrivendevelopment.com/node/617, 2003. Último acceso: 10/02/2008.

[FDD05] Nebulon Pty Ltd. "Feature Driven Development Forums. FDD & Integration Test", http://www.featuredrivendevelopment.com/node/768, 2005.
Último acceso: 10/02/2008.

[FDD06] Nebulon Pty Ltd. "Agile Software Development using Feature Driven Development (FDD)", http://www.nebulon.com/fdd, 2006.
Último acceso: 10/02/2008.

[FDD07] Nebulon Pty Ltd. "The latess FDD Processs",
http://www.nebulon.com/articles/fdd/latestprocesses.html, 2007.
Último acceso: 10/02/2008.

[FG99] Fewster M., Graham, D. "Software Test Automation", ISBN 0-201-33140-3, ACM Press - Addison Wesley, 1999.

[FIT06] Ebe Group, Univ. Calgary, http://ebe.cpsc.ucalgary.ca/ebe, 2006.
Último acceso: 11/02/2008.

[FIT107] Cunningham W. "Framework for Integrated Test",
http://fit.c2.com/wiki.cgi?DownloadNow, 2007. Último acceso: 11/02/2008.

[FIT207] Deng C., Wilson P., Maurer F. "FitClipse: A Fit-based Eclipse Plug-in For Executable Acceptance Test Driven Development", Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007), Come, Italy, 2007.
http://ebe.cpsc.ucalgary.ca/ebe/uploads/Publications/Deng_Wilson_Maurer_FitClipse.pdf. Último acceso: 11/02/2008.

[FIT307] Deng C. "FitClipse: A Testing Tool for Supporting Executable Acceptance Test Driven Development", Master Thesis, Supervisor: Maurer, F., Department of Computer Science, University of Calgary, Calgary, Canada.
http://ebe.cpsc.ucalgary.ca/ebe/uploads/Publications/ChengyaoDengMSc_2007.pdf, 2007. Último acceso: 11/02/2008.

[FM07] Fernández C., Márquez D. "GeneXus Rocha. Episodio Uno", Editorial Grupo Magro, http://www.genexus.com/rocha/Libro_GeneXus_Rocha-Episodio_Uno.pdf, 2007. Último acceso: 11/02/2008.

[Fow05] Fowler M. "The New Methodology",
http://www.martinfowler.com/articles/newMethodology.html, 2005.
Último acceso: 11/02/2008.

[Fow99] Fowler M. et al. "Refactoring: Improving the Design of Existing Code", 1st edition, ISBN 0201485672, Addison-Wesley Professional, 1999.

[Fow107] Fowler M. "C3", http://www.martinfowler.com/bliki/C3.html, 2007.
Último acceso: 11/02/2008.

[Fow207] Fowler M. “Mocks aren’t Stubs”, http://martinfowler.com/articles/mocksArentStubs.html, 01/2007. Último acceso: 11/02/2008.

[Foy06] Foy C. "Fitnesse Selenium Wrapper", http://www.cornetdesign.com/2006/09/fitnesse-selenium-wrapper.html, 2006. Último acceso: 11/02/2008.

[Fre04] Freeman S., Mackinnon T., Pryce N., Walnes, J. “Mock Roles, Not Objects”, OOPSLA’04, Vancouver, Canada, http://www.jmock.org/oopsla2004.pdf, 2004. Último acceso: 31/01/2008.

[Gal07] Gallen B. “Scaling Testing in Scrum”, Agile2007 Conference Presentation, Washington DC, USA, http://www.agile2007.org/agile2007/downloads/presentations/Scaling_Testing_in_Scrum_v1-1_532.pdf, 2007. Último acceso: 11/02/2008.

[Gam94] Gamma E., Helm R., Johnson R., Vlissides J. “Design Patterns: elements of reusable object oriented software”. 1st edition, Addison–Wesley, 1994.

[GBS07] Artech, “GeneXus BPM Suite”, http://www.GeneXus.com/portal/hgxpp001.aspx?2,4,132,O, 2007. Último acceso: 11/02/2008.

[GCW107] GeneXus Community Wiki, “GeneXus Rocha/GXextensions”, http://wiki.gxtechnical.com/commwiki/servlet/hwiki?category%3AGeneXus+Rocha%2FGeneXus+Extensions, 2007. Último acceso: 11/02/2008.

[GCW108] GeneXus Community Wiki, “Knowledge Base Versions”, http://wiki.gxtechnical.com/commwiki/servlet/hwiki?category%3AKnowledge+Base+Versions, 2008. Último acceso: 15/02/2008.

[GCW207] GeneXus Community Wiki, “GeneXus Rocha/GeneXusxtensions/AvailableExtensions”, http://wiki.gxtechnical.com/commwiki/servlet/hwiki?Available+GeneXus+Extensions, 2007. Último acceso: 11/02/2008.

[GCW307] GeneXus Community Wiki, “Bussines Component”, http://www.gxopen.com/commwiki/servlet/hwiki?Business+Components, 2007. Último acceso: 11/02/2008.

[GCW407] GeneXus Community Wiki, “GeneXus Rocha/GeneXus Extensions/Early dopters Program”, http://wiki.gxtechnical.com/commwiki/servlet/hwiki?GeneXus+Extensions+Early+Adopter+Program, 2007. Último acceso: 11/02/2008.

[GCW507] GeneXus Community Wiki, “GeneXus Rocha”, http://wiki.gxtechnical.com/commwiki/servlet/hwiki?category%3AGeneXus+Rocha, 2007. Último acceso: 11/02/2008.

[GCW607] GeneXus Community Wiki, "Patterns - Pattern Instantiation", http://wiki.gxtechnical.com/commwiki/servlet/hwiki?Pattern+Instantiation, 2007. Último acceso: 11/02/2008.

[GCW707] GeneXus Community Wiki, "GeneXus Rocha/Data Selectors", http://wiki.gxtechnical.com/commwiki/servlet/hwiki?category%3AData+Selectors, 2007. Último acceso: 11/02/2008.

[GCW807] GeneXus Community Wiki, *"*What Is a Software Pattern*"*, http://wiki.gxtechnical.com/commwiki/servlet/hwiki?What+Is+a+Software+Pattern, 2007. Último acceso: 11/02/2008.

[GCW907] GeneXus Community Wiki, *"*GeneXus Patterns*"*, http://wiki.gxtechnical.com/commwiki/servlet/hwiki?Patterns, 2007. Último acceso: 11/02/2008.

[GCW1007] GeneXus Community Wiki, "How to Use Data Providers in Other GeneXus Objects", http://wiki.gxtechnical.com/commwiki/servlet/hwiki?How+to+Use+Data+Providers+in+Other+GX+Objects, 2007. Último acceso: 11/02/2008.

[GCW1107] GeneXus Community Wiki, "Data Providers Language Description", http://wiki.gxtechnical.com/commwiki/servlet/hwiki?Data+Providers+Language+Description, 2007. Último acceso: 11/02/2008.

[GCW1207] GeneXus Community Wiki, "Data Providers", http://wiki.gxtechnical.com/commwiki/servlet/hwiki?category%3AData+Provider, 2007. Último acceso: 11/02/2008.

[GDL07] GeneXus Development Library, http://www2.gxtechnical.com/portal/hgxpp001.aspx?15,4,19,O,S,0,MNU;E;25;4;4;1;MNU; , 2007. Último acceso: 11/02/2008.

[GJ03] Gonda B., Jodál N. "GeneXus: Filosofía", http://www.GeneXus.com/portal/hgxpp001.aspx?2,32,660,O,S,0,MNU;E;131;12;MNU; , 2003. Último acceso: 11/02/2008.

[GJ06] Gonda, B., Jodal N. "Evolución de los Objetivos de GeneXus hacias la Cuarta Dimensión", http://www.GeneXus.com/portal/hgxpp001.aspx?2,32,660,O,S,0,MNU;E;131;12;MNU., 2006. Último acceso: 11/02/2008.

[GJ07] Gonda B., Jodal N. "Desarrollo basado en conocimiento: Filosofía y Fundamentos Teóricos de GeneXus", http://www.GeneXus.com/portal/hgxpp001.aspx?2,32,660,O,S,0,MNU;E;131;12;MNU., 2007. Último acceso: 11/02/2008.

[GJ95] Gonda B., Jodal N. “Proyecto GeneXus”. Resumen del trabajo ganador del Premio Nacional de Ingeniería 1995 otorgado por la Academia Nacional de Ingeniería,Uruguay, http://www.aiu.org.uy/gxpsites/agxppdwn?2,1,4,O,S,0,79%3BS%3B1%3B21, , 1995. Último acceso: 11/02/2008.

[Gla02] Glass R. “Facts and Fallacies of Software Engineering”, ISBN: 0-321-11742-5, Addison Wesley, 2002.

[GNU91] Licencia GNU-GPL Version 3, http://www.gnu.org/licenses/gpl.txt, 2007. Último acceso: 11/02/2008.

[GP07] GreenPepper Software, http://greenpeppersoftware.com, 2007. Último acceso: 11/02/2008.

[GQE07] Genexus, “Introducción a GxQuery”, http://www.genexus.com/portal/hgxpp001.aspx?2,7,672,O,S,0,MNU;E;48;13;MNU;, 2007. Último acceso: 11/02/2008.

[GRN07] GeneXus Community Wiki, “GeneXus Rocha Night Builds” http://wiki.gxtechnical.com/commwiki/servlet/hwiki?category%3AGeneXus+Rocha%2FNight+Builds, 2007. Último acceso: 11/02/2008.

[GTW04] Gao J., Tsao J., Wu Y. “Testing and Quality Assurance for Component-Based Software”, ISBN 1580534805, Artech House Publishers, 2003.

[GX07] GeneXus, http://www.GeneXus.com, 2007. Último acceso: 11/02/2008.

[GX08] GeneXus Rocha, “GXTechnical” “GeneXus Rocha”, http://www2.gxtechnical.com/portal/hgxpp001.aspx?15,22,260,O , 2007. Último acceso: 15/02/2008.

[Gxf06] GeneXus Forums, “GeneXus Extensions Forum”, http://www2.gxtechnical.com/portal/hgxpp001.aspx?15,6,40,O,E,0,, 2007. Último acceso: 11/02/2008.

[Gxf07] Genexus BPM Suite, “GxFlow”, http://www.genexus.com/portal/hgxpp001.aspx?2,4,128,O,S,0,MNU;E;37;8;MNU; 2007. Último acceso: 11/02/2008.

[Gxp105] GxPublic 9.0,”Introduction”, http://www.gxopen.com/commwiki/servlet/hwiki?GXpublicYI, 2005. Último acceso: 11/02/2008.

[Gxp205] GxPublic, “Manual del desarrollador”, http://www2.gxtechnical.com/portal/hgxpp001.aspx?15,8,8,O,E,0,,1097, 2005. Último acceso: 11/02/2008.

[GXP07] Genexus, “GxPlorer: Visión General” http://www.genexus.com/portal/hgxpp001.aspx?2,5,673,O,S,0,MNU;E;46;1;MNU; 2007. Último acceso: 11/02/2008.

[Gxu07] Proyecto de Ingeniería de Software “GXUnit”, Docente: Triñanes J. , Facultad de Ingeniería, Universidad de la República, Uruguay, Agosto-Noviembre 2007.

[Gxu106] Almeida E., Araújo A., Larre Borges U. “Proyecto colaborativo GXUnit”, http://www.gxopen.com/commwiki/servlet/hwiki?GXUnit, 2006. Último acceso: 1/12/2007.

[Gxu108] GXUnit1, http://www.gxopen.com/gxopenrocha/servlet/hproject?721 2008. Último acceso 15/03/2008.

[Gxu206] Almeida E. , Araújo A., Larre Borges U. “Collaborative Projects: Experiencias y Testimonios”, “Proyecto Colaborativo GXUnit”, http://www.concepto.com.uy/archivosvinculados/EncuentroGX2006CollaborativeProjects.ppt, 2006. Último acceso 1/12/2007.

[Gxu208] GXUnit2, http://www.gxopen.com/gxopenrocha/servlet/hproject?721 2008. Último acceso 15/03/2008.

[HaC07] Proyecto HamCrest, http://code.google.com/p/hamcrest/wiki/Tutorial, 2007. Último acceso: 11/02/2008.

[Ham04] Hamill, P. “Unit Test FrameWorks”, ISBN 0-596-00689-6, O'Reilly Media Inc, 2004.

[Har05] Harold, E.R. “What’s New in JUnit4”, http://www.cafeaulait.org/slides/sdbestpractices2005/junit4/What's_New_in_JUnit_4.html, 2005. Último acceso: 11/02/2008.

[HOC00] Highsmith J., Orr K., Cockburn, A. “Extreme Programming”, Cutter Consortium’s Agile Project Management Advisory Services White Papers. Cutter Consortium, http://www.cutter.com/content-and-analysis/resource-centers/agile-project-management/sample-our-research/ead0002/ead0002.pdf, 2002. Último acceso: 11/02/2008.

[HT99] Hunt, A., Thomas D. “The Pragmatic Programmer: From Journeyman to Master”, 1st edition, ISBN 0-201-61622-X, Addison Wesley, 1999.

[Hua75] Huang J. “An Approach to Program Testing”, Computing Surveys, Vol. 7, No. 3, Setiembre 1975, http://star.itc.it/ricca/swatII/HUAN75.pdf, Último acceso: 10/02/2008.

[IEEE Std 610.12-1990]
IEEE Standard Glossary of Software Engineering Terminology Institute of Electrical and Electronics Engineers,ISBN: 155937067X, 1990.

[IET99] The Internet Engineering Task Force, “Hypertext Transfer Protocol -- HTTP/1.1”, RFC2616, http://www.ietf.org/rfc/rfc2616.txt, 1999. Último acceso: 11/02/2008.

[ISQ07] International Software Testing Qualifications Board, Certified Tester Foundation Level, Syllabus, Versión 2007. http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf, 05/2007. Último acceso 02/02/2008.

[IXP07] IndustrialXP “Welcome to the Industrial Xp Revolution”,http://industrialxp.org/, 2004. Ültimo acceso 15/03/2008.

[Jai106] Jain N., “Patang Proyect”, http://sourceforge.net/projects/patang, 2006. Último acceso: 11/02/2008.

[Jai206] Jain N. “Integrating Fitnesse with CruiseControl” “Naresh Jain’s WebLob”, http://blogs.agilefaqs.com/2006/07/16/integrating-fitnesse-with-cruisecontrol/, 2006. Último acceso: 11/02/2008.

[Jav07] Sun Microsystems Java, “Java Technology. The Power of Java” http://www.sun.com/java/, 2007. Último acceso: 11/02/2008.

[JavS07] JavaScript, http://developer.mozilla.org/es/docs/JavaScript, 2007. Último acceso: 11/02/2008.

[J2EE] Java Community Process, “JSR 244: Java™ Platform, Enterprise Edition 5 (Java EE 5) Specification”, http://jcp.org/en/jsr/detail?id=244, 2006. Último acceso 11/02/2008.

[Jbh07] JBehave Project, http://jbehave.org/, 2007. Último acceso: 11/02/2008.

[JCP07] Java Community Process, “JSR-000220 Enterprise Java Beans 3.0 (Final Release)”, http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, 2007. Último acceso: 11/02/2008.

[Jef00] Jeffries R., Anderson A., Hendrickson C., Jeffries, R. E., “Extreme Programming Installed”, 1st edition, ISBN 0-201-70842-6 Addison Wesley, 2000.

[Jef04] Jeffries R. “Extreme Programming Adventures in C#”, ISBN 0735619492, Microsoft Press, 2004.

[Jef07] Jeffries R. “Xprogramming.com an Agile Software Development Resource”, http://www.xprogramming.com/, 2007. Último acceso: 11/02/2008.

[JFi05] Hernández N. “JFing: Construcción de una Herramienta CASE para la Automatización del Testing Unitario”, Proyecto de Grado de Ingeniería en Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2005.

[JFi06] Bejar J., Misol M., Nova H. “JFing 2.0: Construcción de una Herramienta CASE para la Automatización del Testing Unitario”, Tutores: Vallespir, D. ,Hernandez, N., Proyecto de Grado de Ingeniería en Computación, Facultad de Ingeniería, Universidad de la República, Uruguay. http://www.fing.edu.uy/inco/grupos/gris/wiki/uploads/Ensenianza/Informe%20Final.pdf, 2005. Último acceso: 20/01/2008.

[JM07] Jeffries R, Melnik G. “Guest Editors' Introduction: TDD--The Art of Fearless Programming “ ,IEEE Computer, Vol.24, Issue 3, pp. 24-30, May/Jun 2007.

[Jmo08] Jmock , http://www.jmock.org/, 2008. Último acceso: 31/01/2008.

[JPZ96] Janicki R., Parnas D., Zucker J., “Tabular Representations in Relational Documents”, Communications Research Laboratory, McMaster University, 1996. http://www.cas.mcmaster.ca/serg/papers/crl313.pdf . Ultimo acceso 03/03/2008.

[Jun107] JUnit Org,, “Resources for Test Driven Development”, http://www.junit.org/, 2007, Último acceso: 01/11/2007.

[Jun207] SourceForge.Net “JUnit: Summary of Changes in version 4.4”, http://junit.sourceforge.net/doc/ReleaseNotes4.4.html, Último acceso: 01/11/2007.

[JuF07] JUnit Factory (experimental service), Agitar Software, http://www.junitfactory.com/, 2007. Último acceso: 30/11/2007.

[Kan00] Kaner C. “Architectures of Test Automation”, *http://www.kaner.com/pdfs/testarch.pdf*, 2000. Último acceso: 16/02/2008.

[Kan02] Kaner C. “Black box software testing: Professional seminar, section 4: The impossibility of complete testing” http://www.testingeducation.org/coursenotes/kaner_cem/cm_200204_blackboxtesting/, 2002. Último acceso: 16/02/2008.

[Kan021] Kaner C. “Black box software testing: Professional seminar, section 19: regression testing” http://www.testingeducation.org/coursenotes/kaner_cem/cm_200204_blackboxtesting/, 2002. Último acceso: 16/02/2008.

[Kan04] Kaner C. “The Ongoing Revolution in Software Testing”, Software Testing & Performance Conference, Florida Institute of Technology, Baltimore, USA 12/8/2004, http://www.kaner.com/pdfs/TheOngoingRevolution.pdf, Último acceso: 16/02/2008.

[KBP01] Kaner C., Bach J., Pretichord B. “Lessons Learned in Software Testing”, ISBN 0471081124, Wiley, 2001.

[Kee02] Keefer G., “Extreme Programming Considered Harmful For Reliable Software Development”, AVOCA GmbH, http://www.agilealliance.org/system/article/file/945/file.pdf, 2002. Último acceso: 20/01/2008.

[Kni07] Kniberg H. “Scrum and Xp from the Trenches. How we do Scrum”, ISBN 978-1-4303-2264-1, C4Media Inc., 2007.

[KL00] Kircher M., Levine D. “The XP of TAO” “eXtreme Programming of Large, Open-source Frameworks”, 1st International Conference on eXtreme Programming and Flexible Processes in Software Engineering, Cagliari, Italy. http://www.cs.wustl.edu/~levine/research/xp2k.ps.gz., 2000. Último acceso: 02/02/2008.

[Koh05] Kolh J. “Conventional Software Testing on a Scrum Team”, http://www.informit.com/articles/article.aspx?p=415981&rl=1, 2005. Último acceso: 02/02/2008.

[Lam07] Lamas J. “Jugo de Extensions”, Presentación sobre las GxExtensions. Tercera reunión de usuarios GeneXus de Montevideo, Uruguay, http://video.google.es/videoplay?docid=7421059744542558069, 2007. Último acceso: 02/02/2008.

[Lar03] Larman C. “Agile and Iterative Development: A Manager's Guide”, ISBN 0-13-111155-8, Addison Wesley, 2003.

[Lar04] Larman C. “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, 3th Edition, ISBN 0-13-148906-2. Addison Wesley Professional, 2004.

[Lbo04] Larre Borges, U. “Metodologías Ágiles para el desarrollo de *Software* – una perspectiva desde el Uruguay”, Tesis de Maestría en Ingeniería en Computación, Director: Brum, F., CPAP, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2004.

[LB03] Larman C., Basili V. “Iterative and Incremental Development: A Brief History”, IEEE Computer; Volume 36, Issue 6, pp 47 – 56, Junio 2003.

[LSLN03] Latorres E., Salvetto P., Larre Borges U., Nogueira, J., “Una herramienta de apoyo a la gestión del proceso de desarrollo de software”, IX Congreso Argentino de Ciencias de la Computación (CACIC 2003), La Plata, Argentina, http://www.latorres.org/enrique/hagdps.pdf, 2003. Último acceso: 19/01/2008.

[LW04] Li K., Wu M “Effective Software Test Automation: Developing an Automated Software Testing Tool”, ISBN 0782143202, SYBEX Inc, 2004.

[MAP01] Manifesto for Agile Software Development, “Principles behind the Agile Manifesto”, http://www.agilemanifesto.org/principles.html, 2001. Último acceso: 19/01/2008.

[Marc05] Marchesi M. “The New XP”, http://www.agilexp.org/downloads/TheNewXP.pdf., 2005. Último acceso: 01/12/2007.

[Mari01] Marick B. “Agile Methods and Agile Testing”, http://www.exampler.com/testing-com/agile/agile-testing-essay.html. Último acceso: 02/02/2008.

[Marq06] Márquez D. “Guia Práctica GeneXus. Desarrollo Basado en el Conocimiento”, ISBN 9974799007, Editorial Grupo Magro, 2006.

[Mar07] Martin R.C. “Professionalism and Test-Driven Development”, IEEE Computer; Volume 24, Issue 3, pp 32 - 36, May-Jun 2007.

[MA01] Manifesto for Agile Software Development, http://www.agilemanifesto.org, 2001. Último acceso: 19/01/2008.

[MC05] Mugridge R., Cunningham W. ”Fit for Developing Software: Framework for Integrated Tests”, ISBN 0-321-26934-9, Prentice Hall PTR, 2005.

[Mel07] Melnik G. “Empirical Analyses of Executable Acceptance Test Driven Development”, PHD Thesis, Supervisor: Maurer, F., Departament of Computer Science, University of Calgary, Calgary, Alberta, http://ebe.cpsc.ucalgary.ca/ebe/uploads/Publications/MelnikPhD.pdf, 2007. Último acceso: 11/02/2008.

[Mes07] Meszaros G. “xUnit Test Patterns.Refactoring Test Code”, ISBN 0-13-149505-4, Addison-Wesley, 2007.

[MFC01] Mackinnon T., Freeman S., Craig P. (Independent) “Endo-Testing: Unit Testing with Mock Objects”, Conferencia “eXtreme Programming and Flexible Processes in Software Engineering“, XP2000, Cerdeña, Italia, http://www.mockobjects.com/files/endotesting.pdf, 2000. Último acceso: 29/01/2008.

[MH04] Massol V., Husted T. “JUnit in Action”, ISBN 1-930110-99-5, Manning Publications Co., 2004.

[Mil07] Miller J. “Is there a future for Fit testing?” http://codebetter.com/blogs/jeremy.miller/archive/2007/07/30/is-there-a-future-for-fit-testing.aspx, 2007. Último acceso: 29/01/2008.

[Mil106] Miller J "Create a Testing DSL with FitNesse and Selenium", http://codebetter.com/blogs/jeremy.miller/archive/2006/07/15/147400.aspx, 2006. Último acceso: 29/01/2008.

[Mil206] Miller J. “Tigris.org Open Source Software Engineering Tools” “Storyteller: Automated acceptance testing for .Net”, http://storyteller.tigris.org/, 2006. Último acceso: 11/02/2008.

[MMW07] Martin R., Martin M., Wilson P. "Welcome to FitNesse", http://www.fitnesse.org, 2007. Último acceso: 11/02/2008.

[MM07] Melnik G., Maurer F. “Multiple Perspectives on Executable Acceptance Test-Driven Development”, Department of Computer Science, University of Calgary, Canada, http://ebe.cpsc.ucalgary.ca/ebe/uploads/Publications/XP2007_Melnik_Maurer.pdf, 2007. Último acceso: 11/02/2008.

[Moz07] Mozzilla “Firefox 2”, http://www.mozilla.com/en-US/firefox/, 2007. Último acceso: 11/02/2008.

[MRM04] Melnik G., Read K., Maurer, F. “Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective”, Department of Computer Science, University of Calgary, Canada, http://ebe.cpsc.ucalgary.ca/ebe/uploads/Publications/MelnikReadMaurer2004b.pdf, 2004. Último acceso: 11/02/2008.

[MS01] McGregor J. ,Sykes, D. “A Practical Guide to Testing Object-Oriented Software”, ISBN 0-201-32564-0, Addison-Wesley, 2001.

[Mug03] Mugridge R. “Test Driven Development and the Scientific Method”, Department of Computer Science,University of Auckland,New Zealand, http://agile.openmex.com/agile2007/2003/files/P6Paper.pdf, 2006, Último acceso: 11/02/2008.

[Mug107] Mugridge R. “Fitlibrary” http://sourceforge.net/projects/fitlibrary , 2007. Último acceso: 11/02/2008.

[Mug207] Mugridge R. “Welcome to Zibreve”, http://www.zibreve.com, 2007. Último acceso: 11/02/2008.

[MUT07] Microsoft Corporation, “Microsoft Research: Project MUTT”, http://research.microsoft.com/projects/mutt/ ,2007. Último acceso: 11/02/2008.

[Mye04] Myers G. “The art of software testing, 2nd edition”, ISBN 0-471-46912-2, John Wiley & Sons Inc., 2004.

[MW78] Manna Z. ,Waldinger R. “The Logic of Computer Programming”, IEEE *Software*, Vol.4, Issue 3, pp. 199-229, Mayo 1978.

[Net07] Microsoft .Net Framework, http://www.microsoft.com/spanish/msdn/latam/netframework/, 2007 Último acceso: 19/4/2008.

[New04] Newton S. “AntFit”, http://www.cmdev.com/antfit/docs/ , 2004. Último acceso: 11/02/2008.

[Nor06] North D. “Introducing BDD”, http://dannorth.net/introducing-bdd/, 2006. Último acceso: 11/02/2008.

[Nun07] NUnit, http://www.nunit.org/, 2007, Último acceso 31/01/2008.

[Ols07] Olsen K. "Using Scrum as a Test Management method", International SW Testing Conference (ASTA), Tutorials, Seul, Korea, www.softwaretest.dk/Nonsec/webselv/softwaretest/docs/1/Scrum_presentation_Klaus_Olsen_v1.0.pdf, 2007. Último acceso: 11/02/2008.

[OST07] OpenSourceTesting. "Open Source Software Testing Tools, news and discussion", Publisher Aberdur M., http://www.opensourcetesting.org/, 2007. Último acceso: 11/02/2008.

[Per04] Pérez B. "GXP – Adaptación y Aplicación de eXtreme Programming", Serie reportes técnicos, ISSN: 0797–6410, RT0417, Grupo de Ingeniería de Software (Gris), Instituto de Computación, Facultad de Ingeniería, Pedeciba Área Informática, Universidad de la República, Uruguay. http://www.fing.edu.uy/inco/pedeciba/bibliote/reptec/TR0417.pdf, 2004. Último acceso 20/12/2007.

[Per06] Pérez B. "Proceso de Testing Funcional Independiente", Tesis de Maestría en Informática, Directora: Szasz, N., PEDECIBA Informática, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2006.

[Pet102] Pettichord B. "Testers Should Embrace Agile Programming", http://www.io.com/%7Ewazmo/papers/embrace_agile_programming.html, 2002. Último acceso 04/02/2008.

[Pet104] Pettichord B. "Homebrew Test Automation", http://www.io.com/%7Ewazmo/papers/homebrew_test_automation_200409.pdf Último acceso 04/02/2008.

[Pet202] Pettichord B. "Agile Testing What is it? Can it work?", http://www.io.com/~wazmo/papers/agile_testing_20021015.pdf, 2002. Último acceso: 04/02/2008.

[Pet204] Pettichord B. "Agile Testing Challenges", Pacific Northwest Software Quality Conference, Portland, Oregon,October 2004, http://www.io.com/%7Ewazmo/papers/agile_testing_challenges.pdf Último acceso 04/02/2008.

[PEX07] Microsoft Corporation, Project Pex: Program Exploration, "Microsoft Research" "Pex: Dynamic Analysis and Test Generation for .NET", http://research.microsoft.com/Pex/, 2007. Último acceso 04/02/2008.

[Pop07] Proyecto Popper, http://popper.tigris.org/, 2007. Último acceso 04/02/2008.

[RBG98] Ryser J., Berner S., Glinz M."On the State of the Art in Requirements-based Validation and Test of Software", Technical Report 98-12, Departament of Computer Science of the University of Zurich, Zurich, Switzerland, ftp://ftp.ifi.unizh.ch/pub/techreports/TR-98/ifi-98.12.pdf, 1998. Último acceso: 11/02/2008.

[Rea05] Read K. “Supporting Agile Teams of Teams via Test Driven Design”, Master Thesis, Departament of Computer Science, University of Calgary, Calgary, Canada, http://ebe.cpsc.ucalgary.ca/ebe/uploads/Publications/Read2005.pdf, 2005. Último acceso: 11/02/2008.

[Rsp05] RSpec, http://rspec.rubyforge.org/, 2007. Último acceso: 11/02/2008.

[RS05] Rainsberger J., Stirling S. “JUnit Recipes Practical Methods for Programmer Testing”, ISBN 1932394230, Manning Publications Co. , 2005.

[Saf07] Saff D. “Theory-infected Or How I Learned to Stop Worrying and Love Universal Quantification”, OOPSLA’07, Montreal, Quebec, Canada., http://groups.csail.mit.edu/pag/pubs/test-theory-demo-oopsla2007.pdf, 2007. Último acceso: 11/02/2008.

[Sal06] Salvetto P. “Modelos Automatizables de Estimación muy Temprana del Tiempo y Esfuerzo de Desarrollo de Sistemas de Información”, Tesis de Doctorado, Directores: Segovia, F. , Nogueira J., Facultad de Informática, Universidad Politécnica de Madrid, Doctorado conjunto en ingeniería informática UPM-ORT, http://oa.upm.es/367/01/PEDRO_SALVETTO_LEON.pdf, 2006, Último acceso: 11/02/2008.

[SB01] Schwaber K., Beedle M. “Agile Software Development with SCRUM”, ISBN 0130676349, Prentice Hall, 2001.

[SB07] Saft D., Boshernitsan M. “The Practice of Theories: Adding “For-all” Statements to “There-Exists” Tests”, http://shareandenjoy.saff.net/tdd-specifications.pdf, 2007. Último acceso: 11/02/2008.

[Sch95] Schwaber K. “Scrum development process”, Business Object Design and Implementation Workshop, OOPSLA’95, Austin, Texas, USA, http://jeffsutherland.com/oopsla/schwapub.pdf, 1995. Último acceso: 1/02/2008.

[Scr07] Scrum “Scrum, It’s about common sense”, http:\\www.controlchaos.com, 2007. Último acceso: 11/02/2008.

[Sel06] OpenQa “OpenQa: Selenium”, http://www.openqa.org/selenium/, 2006. Último acceso: 11/02/2008.

[Sho07] Shores J. “James Shores Successfull Software” “Five Ways to Misuse FIT”, http://www.jamesshore.com/Blog/Five-Ways-to-Misuse-Fit.html, 2007. Último acceso: 11/02/2008.

[Sim95] Simonyi C. "The Death Of Computer Languages, The Birth of Intentional Programming", Technical Report MSR-TR-95-52, ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.doc, 1995. Último acceso: 11/02/2008.

[Sin06] Siniaalto M. "Test driven development: empirical body of evidence", "Agile Software Development of Embedded Systems", Versión 1.0, ITEA, Agile VTT, http://www.agile-itea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf, 2006. Último acceso: 31/01/2008.

[SØ05] Skytteren S.K., Øvstetun T.M. "AutAT:tool for automatic acceptance testing", Master Thesis, Departament of Computer Science and Information Science, Faculty of Information Technology, Mathematics and Electrical Engineering, Norwegian University of Science and Technology, http://boss.bekk.no/boss/autat/docs/report.pdf, 2005. Último acceso: 30/11/2007.

[SP01] Schuh P., Punke S. "ObjectMother, Easing Test Object Creation in XP", http://www.xpuniverse.com/2001/pdfs/Testing03.pdf, 2001. Último acceso: 31/01/2008.

[SR03] Stephens M. ,Rosenberg D "Extreme Programming Refactored: The Case Against XP", 1st edition, ISBN 1-590-59096-1, Addison Wesley, 2003.

[Sti06] SolutionsIQ "Story Test Iq", http://storytestiq.sourceforge.net/ 2006. Último acceso: 31/01/2008.

[Sto05] Stott W. "Get Your Customers Involved in the Testing Process with Functional Tests in Excel", http://msdn.microsoft.com/msdnmag/issues/05/02/Excel UnitTests/default.aspx, 2005. Último acceso:11/02/2008.

[SWE04] Guide to the Software Engineering Body of Knowledge SWEBOK, 2004 version. IEEE, Computer Society. http://www.swebok.org, 2004. Último acceso:11/02/2008.

[Tas02] Tassey G. "The Economic Impacts of Inadequate Infrastructure of Software Testing", Final Report, NIST National Institute of Standard & Technology, http://www.nist.gov/director/prog-ofc/report02-3.pdf, 2002. Último acceso: 11/02/2008.

[TN86] Takeuchi H, Nonaka I. "The New New Development Game", Hardvard Bussiness Review, January-February 1986.

[Tos07] Toshiaki D. "Uso eficaz de métricas em métodos ageis de desenvolvimento e software", Tesis de maestría, Orientador: Goldman, A., Instituto de Matemática y Estadística, Univ. de San Pablo, Brasil, , Agosto 2007. http://grenoble.ime.usp.br/~gold/orientados/dissertacaoDaniloSato.pdf. Último acceso 20/11/2007.

[TS06] Tillmann N., Schulte W. "Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution", Microsoft Technical Report MSR-TR-2005-153, Microsoft Research, Microsoft Corporation. http://research.microsoft.com/~schulte/Papers/UnitTestsReloaded(MSR-TR-05-153).pdf, 2006.

[TSG05] Tillmann N., Schulte W., Grieskamp G. "Parameterized unit tests", Microsoft Technical Report MSR-TR-2005-64, Microsoft Research, Microsoft Corporation, ftp://ftp.research.microsoft.com/pub/tr/TR-2005-64.pdf, 2005. Último acceso: 11/02/2008.

[Val06] Vallespir D. "Mejora de la Calidad de los Prototipos Desarrollados en un Contexto Académicos", Tesis de Maestría en Informática, Director: Ruggia R., Supervisor: Tasistro, A., PEDECIBA Informática, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2006.

[Vet06] Vettrivel V. "FIT and Eclipse: Testing with the Extended FIT Eclipse plug-in", http://www-128.ibm.com/developerworks/aix/library/au-fiteclipse/index.html#fig1, 2006. Último acceso: 11/02/2008.

[Vla06] Vlagsma P. "FitBook Examples ported to .NET 2.0", http://www.vlagsma.com/fitnesse/Default.aspx, 2006. Último acceso: 11/02/2008.

[Wal05] Walnes J. "Flexible JUnit assertions with assertThat()", http://joe.truemesh.com/blog/000511.html, 2005. Último acceso: 11/02/2008.

[Wat02] Waterfall Model, http://en.wikipedia.org/wiki/Waterfall_model, 2008. Último acceso: 11/02/2008.

[Web20] O'Reilly Tim "What Is Web 2.0" "Design Patterns and Business Models for the Next Generation of Software", 30/09/2005, http://http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html. Último acceso 05/05/08.

[Wel06] Wells D. "Extreme Programming: A gentle introduction", http://www.extremeprogramming.org. 2006. Último acceso: 11/02/2008.

[Whi00] Whittaker J. "What is Software Testing? And Why Is It So Hard?", IEEE Software, Vol. 17, No. 1, pp. 70-79, Enero 2000.

[Wie06] Wiesner S. "FitNesse & Test Coverage", http://fitnesse.testmanager.info/fitnesse/FitNesse_Testcoverage.pdf, 2006. Último acceso: 11/02/2008.

[Wil05] Williams L. "Case Study Retrospective: Kent Beck's XP Versions 1 and 2", Ponencia en "Annual Research Review 2005", Center for Software Engineering, North Carolina State University, http://sunset.usc.edu/events/2005/arr/proceedings/presentations/OneDayWorkshops/agile/williams.pdf , 2005. Último acceso: 11/02/2008.

[WV05] Wodzislawski M., Vázquez G. "El desafío del testing: ¿Artesanía o Metodología?" – Conferencia en el XV Encuentro de Usuarios GeneXus, 2005.

[W3C04] World Wide Web Consortium (W3C) "*Web Services Architecture*", http://www.w3.org/TR/ws-arch/, 2004. Último acceso: 11/02/2008.

[W3C107] World Wide Web Consortium (W3C), "XHTML2 Working Group Home Page", http://www.w3.org/MarkUp/, 2007. Último acceso: 11/02/2008.

[XML06] World Wide Web Consortium "W3C" "Extensible Markup Language (XML) 1.0 (Fourth Edition)", http://www.w3.org/TR/2006/REC-xml-20060816, 2006. Último acceso: 11/02/2008.

[XPG07] Xpoogle http://www.xpoogle.com , 2007. Último acceso: 11/02/2008.

Glosario

Build — En el desarrollo con GeneXus se entiende por ***build*** al proceso de creación del código en lenguaje de destino de las aplicaciones.

Especificar — En el desarrollo con GeneXus se entiende por **especificar** a la generación de un archivo de especificación por cada objeto que describe el comportamiento del objeto. Se utiliza en la especificación un lenguaje intermediario que es independiente del lenguaje objetivo de la aplicación [Art205].

Generar — En el desarrollo con GeneXus se entiende por **generar** al proceso que genera el código fuente en el lenguaje de destino. Se hace en base al contenido de los archivos de especificación [Art205].

Navegación — En el desarrollo con GeneXus se entiende por **navegación** el recorrido que se efectuará para acceder a los datos, incluyendo el plan de acceso, las condiciones de filtro de los mismos y el conjunto de atributos a entregar [Art205].

Marco (*FrameWork*) — Aplicación semi completa que provee de una estructura común y reutilizable que puede ser compartida entre varias aplicaciones. Los desarrolladores pueden incorporar el marco y extenderlo para resolver sus necesidades específicas [MH04].

A n e x o A

El proyecto GXUnit

El objetivo del proyecto GXUnit es la construcción de una herramienta para automatización de pruebas unitarias asociada a GeneXus. En este capítulo se expone acerca de dicho proyecto y de los productos resultantes de dos procesos de desarrollo independientes y paralelos que dieron lugar a las versiones iniciales de la herramienta, llevados adelante por dos grupos de estudiantes del curso "Proyecto de Ingeniería de *Software*", de la Facultad de Ingeniería de la Universidad de la República. Dichas versiones se distinguirán con las denominaciones GXUnit1 y GXUnit2.

Este capítulo está organizado de la siguiente manera:

- En la sección A.1 se ofrece una reseña del proyecto GXUnit y se explican características generales.
- En la sección A.2 se describe la herramienta GXUnit1.
- En la sección A.3 se describe la herramienta GXUnit2.

Se desea destacar que las secciones A.2 y A.3 de este capítulo contienen un resumen elaborado a partir de la documentación producida por ambos grupos de estudiantes, la cual expone los resultados de su investigación y de su trabajo en respuesta a los requerimientos planteados por los Clientes.

A.1 Reseña

En esta sección se brindan antecedentes acerca del proyecto, los requerimientos suministrados a ambos equipos de desarrollo y un resumen de las principales características de las herramientas producidas, denominadas GXUnit1 y GXUnit2.

A.1.1 Antecedentes

La propuesta GXUnit tuvo su origen en el año 2003 cuando Enrique Almeida [Alm03] anunció su objetivo de crear una herramienta de automatización de pruebas unitarias en GeneXus. En el año 2004 formalizó su propuesta [Alm04] caracterizando la misma en la línea de las herramientas xUnit. Planteó que los objetivos básicos a cumplir serían los siguientes:

- Crear un marco de pruebas asociado a GeneXus.
- Poder escribir las pruebas en GeneXus, bajo la forma de *procedures.*
- Ejecutar las pruebas y registrar los resultados.

Considerando para el desarrollo un enfoque iterativo e incremental postuló ir cumpliendo con el objetivo de generar objetos GeneXus para pruebas de:

- Objetos sin UI (*Procedures*)
- *WebPanels* (pantallas de consulta web)
- *Transacciones (Transactions)*
- *WorkPanels* (pantallas de consulta win)

En el año 2006 se retoma bajo la forma de “Proyecto Colaborativo”, al cual nos integramos junto con Uruguay Larre Borges. Se definieron como objetivos elaborar una especificación funcional preliminar, abarcando la prueba de *procedures*, y estudiar opciones para viabilizar su programación. En [Gxu206] y [Gxu106] se expuso acerca de la experiencia y se publicó una aproximación a su especificación funcional.

En julio de 2007 se propuso el proyecto GXUnit para su desarrollo en el ámbito del curso “Proyecto de Ingeniería de *Software*” de la Facultad de Ingeniería de la Universidad de la República. Se acordaron con el resto de los proponentes del proyecto los requerimientos mínimos a cumplir, incluyéndose funcionalidad relativa a la parametrización de las pruebas mediante datos externos contenidos en tablas (propuesta en la contribución principal de este trabajo), como un requerimiento para la definición de las pruebas; oficiando de prueba de concepto. Otro importante requerimiento fue la generación de código GeneXus para pruebas, de forma de permitir la portabilidad.

Entre agosto y noviembre del año 2007 dos grupos de estudiantes desarrollaron en forma independiente y en paralelo, dos versiones iniciales de la herramienta GXUnit

en el contexto del mencionado curso. Se actuó en el rol de Cliente, realizándose el seguimiento de la construcción y validación de los prototipos, junto al resto de los proponentes.

Los productos de *software* resultantes pueden ser descargados desde Internet [Gxu108] [Gxu208].

A.1.2 Requerimientos

Se suministraron a ambos grupos la siguiente misión y los siguientes requerimientos. El alcance se negoció y acotó con cada uno de los grupos.

Misión: Construir una herramienta que permita automatizar las pruebas unitarias de objetos GeneXus a ser utilizada por Analistas GeneXus.

Consideraciones técnicas:

La herramienta se construirá como una extensión para la versión de GeneXus en elaboración denominada "Rocha". Deberá tener un diseño modular y presentar sus fuentes "*código limpio que funcione*". Es preciso tener muy en cuenta que esta es una etapa inicial de la herramienta, que irá desarrollándose en incrementos. A futuro se espera poder incorporarle componentes que permitan generar pruebas para otros objetos GeneXus que no serán contemplados en esta primera edición y nuevos editores para los casos de prueba. El reuso, la independencia entre capas y un bajo acoplamiento son requisitos importantes. Se deberá programar en C# y se utilizará el SDK suministrado por Artech para la programación de extensiones.

Relatos:

- El Analista definirá "casos de prueba" desde el IDE de GeneXus, desde donde también podrá comandar su ejecución y ver los resultados. Se deberá suministrarle un editor para efectuar el ingreso de estos casos de prueba.
- El Analista podrá ejecutar un conjunto de pruebas en modo desatendido, por ejemplo luego de una construcción (*build*), adicionalmente a la ejecuciones que realiza desde la interfase.
- Los casos de prueba se almacenarán en la KB del SUT.
- El resultado de la ejecución de las pruebas no se almacenará en la KB.
- Se desea que la herramienta tenga una interfase similar a los productos xUnit para la ejecución y visualización de primer nivel de los resultados. Se desea también que esté acorde a la interfase del IDE de GeneXus, para que no resulte extraña al Analista.
- Se definen como objetos verificables aquellos pertenecientes al dominio de objetos para los cuales GXUnit puede generar y ejecutar pruebas unitarias. En esta primera instancia estos serán solamente objetos de tipo *procedure*.

- La herramienta generará en forma automatizada objetos verificadores de tipo *procedure* (*tests*) a partir de los objetos verificables y de los casos de prueba. Serán los encargados de guiar la prueba invocando a los objetos a verificar.
- El Analista necesita poder conocer rápidamente cuales son los objetos verificables y cuales de estos tienen casos de prueba ya definidos.
- Se deberá implementar un oráculo que permitirá saber si cada prueba fue exitosa o fallida. El Analista podrá indicar que se realice una comprobación parcial, lo cual implica que habrá resultados de la prueba en este caso que no influirán en la decisión acerca de si la misma pasa o falla.
- Las pruebas fallidas se marcarán en Rojo, las que pasaron en Verde, aquellas no ejecutados en Gris. De atraparse excepciones estas se desplegarán pintadas de Amarillo.
- Existirán procedimientos "verificadores del usuario" (PVU), que escribirá el Analista, y que podrán ser asociados a las pruebas para ser invocados previo a su finalización. Estos procedimientos implementarán oráculos y poseerán un único parámetro "*booleano*" para indicar si la prueba fue exitosa o fallida.
- El Analista definirá una prueba indicando:
  - o El objeto verificable.
  - o Uno o varios PVUs.
  - o Un conjunto de datos ("tuplas o casos de prueba") de entrada y resultados esperados que deberán corresponderse respectivamente con cada uno de los parámetros de entrada y de salida del objeto verificable.
- El editor del caso de prueba deberá presentar una grilla "inferida" a partir de los parámetros de entrada y salida del objeto verificable para que el Analista ingrese o modifique los casos de prueba (datos de entrada y resultados esperados).
- Para un mismo objeto verificable se pueden definir varios "casos de prueba". El Analista podrá indicar cuales de estos casos desea incluir en la próxima prueba y cuales desea ignorar.
- Durante la ejecución de una prueba se iterará la invocación del objeto a verificar, una vez por cada "caso de prueba" participante en la prueba, pasándole como parámetros de entrada los datos de entrada indicados en un "caso de prueba". Al finalizar la prueba se invoca al PVU (de existir).
- Dada una prueba, el Analista deberá poder visualizar, para cada "caso de prueba" los datos de entrada, los resultados esperados y datos de salida obtenidos. También necesitará acceder al resultado de ejecuciones anteriores.
- Una prueba se considera fallida si produce resultados no coindicentes con los esperados y/o si el PVU indica falla. La comparación de los resultados esperados con los obtenidos, para el alcance de este desarrollo, se limitará a la igualdad.
- Debe ser posible, para un Analista, cargar los datos de entrada y resultados esperados, desde un archivo externo.
- Los tipos de datos a considerar son los básicos de GeneXus, incluyendo SDT.

- El resultado de la ejecución de las pruebas se podrá almacenar con diferentes niveles de detalle.
- Los cambios que se produzcan a nivel de los parámetros de un objeto verificable deberán impactarse contra su(s) caso(s) de prueba, detectándose automáticamente las diferencias para permitir tanto su reconstrucción automatizada como la de los procedimientos verificadores (*tests*).
- La eliminación de PVUs implicará advertir al Analista acerca de los casos de prueba que los utilicen y reconstruir en forma automatizada los casos de prueba y los procedimientos verificadores (*tests*).
- Debe existir un mecanismo para eliminar fácilmente un objeto verificable con casos de prueba asociados. Dicha eliminación también eliminará sus casos de prueba y procedimientos verificadores.
- Al Analista le interesará verificar el estado de la base de datos luego de la ejecución de una prueba, por lo cual se desea proveer un mecanismo en tal sentido que le permita conocer la variación n BD = BDinicial - BDfinal y verificar que el estado final sea el esperado.

A.1.3 Principales características

Se obtuvieron como resultantes principales del proyecto dos productos de *software*, que se denominan como GXUnit1 y GXUnit2. El diseño de la solución estuvo a cargo de cada uno de los grupos de estudiantes. Junto a los productos de *software* produjeron documentos de análisis, diseño, pruebas, manuales de usuario, y una guía para la continuación del desarrollo. Dicha documentación técnica se resumirá en las secciones A.2 y A.3.

A partir del mismo conjunto de requerimientos cada grupo desarrolló una herramienta con diferencias importantes en cuanto al diseño de la solución.

Las semejanzas están dadas por haber satisfecho un núcleo común de requerimientos y también por algunas soluciones de diseño común, como ser el almacenamiento de los resultados de las pruebas en formato XML y la utilización de *Web Services* para comandar su ejecución desde la extensión. Las diferencias surgen en el diseño de las interfases, su vinculación con los objetos GeneXus, la generación de procedimientos GeneXus para prueba y el almacenamiento de los resultados. También surgen diferencias debidas al alcance acordado con cada grupo de estudiantes.

Se desean destacar las siguientes características:

GXUnit1:

- **Genera un objeto verificador por cada objeto verificable** Estos procedimientos verificadores pueblan la KB y se corresponden uno a uno con la prueba y el objeto a verificar. No presentan restricciones en la cantidad de parámetros a utilizar.

- **El objeto verificador se implementa como un *Web Service*** que es consumido desde la extensión
- **Crea una Parte nueva para todo procedimiento** En dicha parte se muestra el editor del caso de prueba, en el cual pueden incluirse las "tuplas o casos de prueba" de valores de entrada y de salida.
- **Almacena los datos para prueba y los resultados** en archivos XML.
- **Ofrece una primera aproximación a la verificación de la BD.**
- **Admite procedimientos con parámetros SDT.**
- **Permite reconstruir un caso de prueba ante cambios en la regla parm del procedimiento a verificar** minimizando la perdida de datos.

GXUnit2:

- **Genera un único objeto verificador, capaz de ejecutar todas las pruebas, pero en esta implementación solo para procedimientos con dos parámetros de entrada y uno de salida.** Dicho objeto utiliza invocaciones dinámicas[81] para ejecutar los procedimientos a probar pero necesita también de la posibilidad de pasar parámetros definidos en forma dinámica. Dado que esto último no es posible hacerlo aún en GeneXus pero existe la posibilidad que se implemente a futuro, se resolvió, para poder construir un prototipo de la solución, generar pruebas solo para procedimientos con dos parámetros de entrada y uno de salida.
- **El objeto verificador se implementa como un *Web Service*** que es consumido desde la extensión
- **Crea un Objeto nuevo en la KB denominado "TestSet":** Dicho objeto permite definir los casos de prueba para un objeto a verificar. Tiene un editor que permite ingresar las "tuplas o casos de prueba" con datos de entrada y salidas esperadas así como los procedimientos verificables y comentarios.
- **Almacena los datos para pruebas** en la base de datos en que reposa la KB, mientras que los resultados son almacenados en archivos XML.
- **Permite reconstruir un caso de prueba ante cambios en la regla parm del procedimiento a verificar** minimizando la perdida de datos.
- **Genera un log a diferentes niveles.**

[81] En GeneXus es posible indicar en una invocación (p/ej. en un *Call*) que el nombre del objeto a invocar se obtenga a partir del contenido de una variable, de forma que en tiempo de ejecución pueda invocarse a diferentes objetos. Pero estos objetos deben tener los mismos parámetros, en caso contrario fracasaría la invocación en tiempo de ejecución, produciendo una excepción. Esta limitante sería levantada en futuras versiones de GeneXus, quedando a estudio de Artech.

A.2 GXUnit1

En esta sección se resume la documentación del producto GXUnit1.

A.2.1 Documentación técnica sobre GeneXus

En esta subsección se introduce a la arquitectura de Genexus Rocha y al modelo de clases obtenido a partir de la investigación realizada.

A.2.1.1 Interfase de Extensiones de GeneXus[82]

GeneXus Rocha presenta una arquitectura en 3 capas, como muestra la figura A-1, *User Interface, Business Logic y Data Layer*.

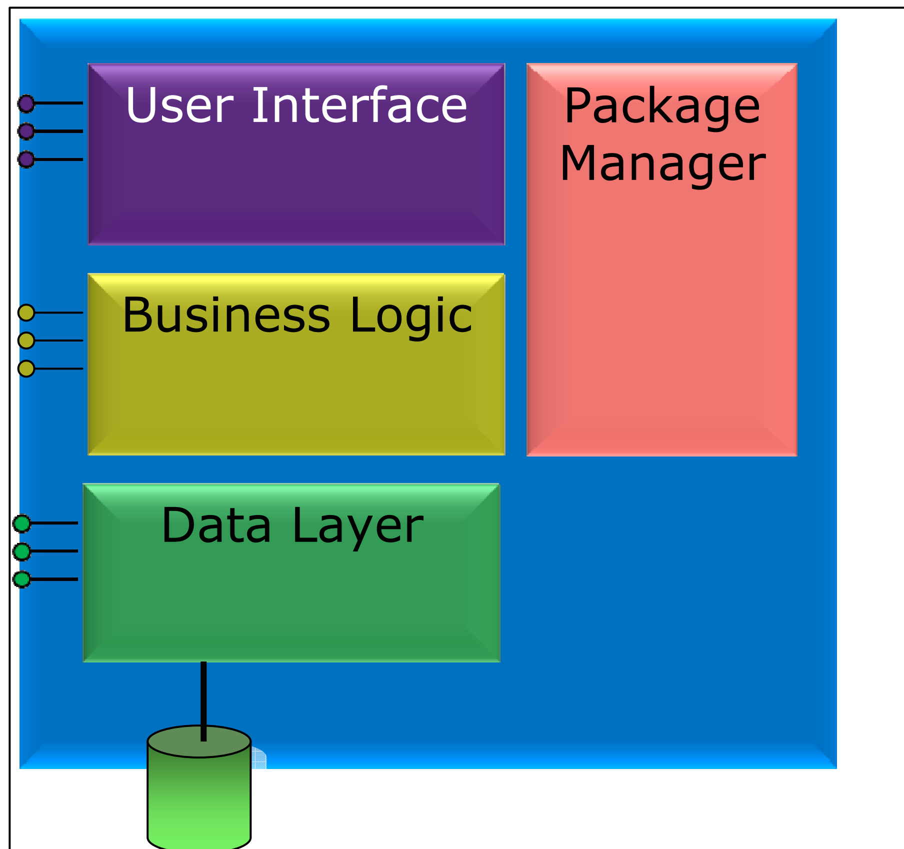


Fig. A- 1: Arquitectura de GeneXus Rocha (gxUnit1).

La capa *Business Logic* está encargada de manejar la estructura básica, *Knowledge Base*, objetos GX, partes, etc. La extensibilidad de este diseño fue lograda mediante un esquema de paquetes y servicios:

[82] Según se cita en el documento "Documentación Tënica" de GXUnit1 el orígen de esta información se situa en una conferencia sobre GXextensions brindada en el Encuentro de Usuarios GeneXus de Setiembre de 2007.

- **Servicios UI:** KB, *Command*, Menu, *ToolWindows*, Documentos, Editores, Propiedades, Selección, *Drag&Drop*, etc.
- **Servicios GX UI:** Selección *Att/Var*, Objetos, Imágenes; Armado, *Inspectors*.
- **Servicios Data Layer:** *CopyModel.*
- **Servicios Business Logic:** *KnowledgeManager, Search/Index*
- **Servicios GX BL:** Normalización, Ejecución, InfoTablas/TRN, *UserControlManager*

A.2.1.2 Modelo de clases

La estructura de clases que se obtuvo a partir de las primeras etapas de investigación con el SDK de extensiones se ofrece en el siguiente diagrama:

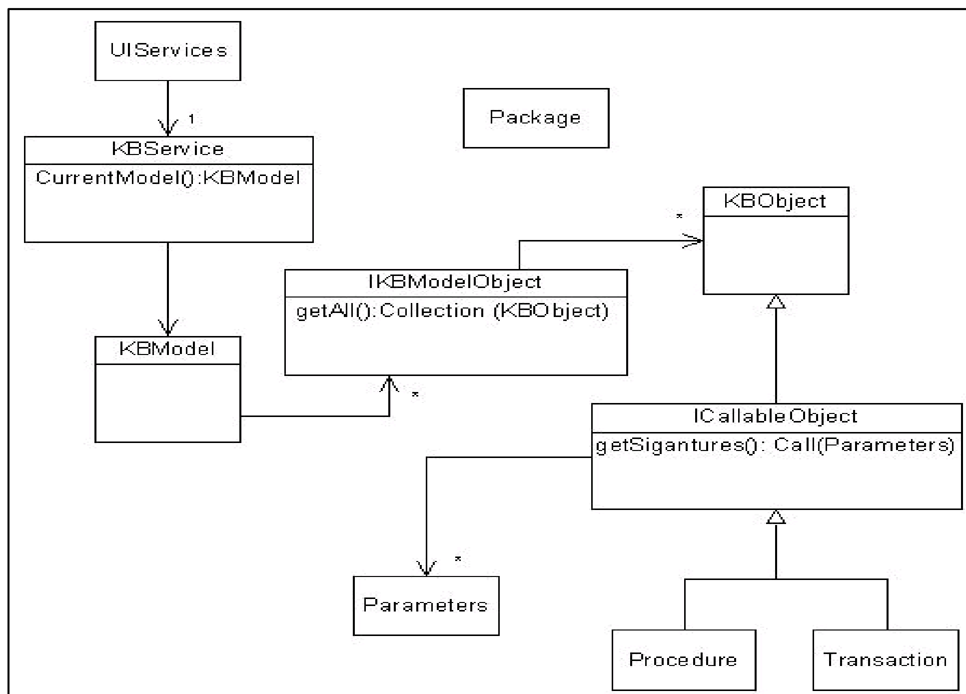


Fig. A- 2: Estructura de clases según la investigación para GXUnit1.

A.2.2 Características de la solución técnica

En esta subsección se resumirá acerca de la solución encontrada al problema en cuestión utilizando la interfase de extensiones de GeneXus (de ahora en adelante GX).

Verificación de Objetos Procedimiento GX
La idea general utilizada para la resolución de este problema es la generación de un nuevo objeto procedimiento (*procedure*) GX. Para ello, y utilizando los servicios de la interfase de extensiones de GX, se analizó la lista de parámetros declarados en el objeto procedimiento a probar, así como su tipo y si es de entrada y/o salida.

Editor de casos de prueba
En base a los datos recabados sobre los parámetros del objeto verificable se genera un editor que permite ingresar casos de prueba para el procedimiento, especificando datos para los parámetros de entrada y datos para los valores esperados de los parámetros de salida. Este ingreso se realiza en una parte, que es seleccionada por intermedio de una pestaña denominada *"test"* en los objetos verificables.

Formato para almacenar los datos de prueba y los resultados
Los datos de prueba y los resultados se almacenan en archivos con formato XML.

Generación del Procedimiento GX verificador
De la misma manera que con el Editor de casos de prueba, para generar el procedimiento GX verificador se recaban datos sobre los parámetros del objeto verificable, y se genera código GX acorde para la carga de datos e invocación, mediante una llamada *Call* de GX, del procedimiento verificable. Se utilizaron los tipos de variables GX *"XMLReader"*, "*XMLWriter"*, y "*File"* para manejar desde el código GX la lectura de los casos de prueba y la escritura de los resultados.

Invocación del procedimiento GX verificador
El procedimiento verificador se generó con la propiedad "*Main Program*" en verdadero y la invocación es realizada a través del protocolo HTTP [IET99]. Esta invocación puede ser realizada de forma manual por el usuario – a través de la interfase de ejecución de pruebas – o en forma automática, como proceso por lotes al finalizar el proceso de *Build* (mediante suscripción al evento *AfterComplete*). La invocación del procedimiento verificable genera un archivo XML de resultados.

Interfase de visualización de resultados de pruebas
Se implementó una sencilla interfase gráfica para visualizar los resultados de las pruebas, utilizando un componente *TreeView* para cargar los nodos del XML. Además se utilizó el código de colores Verde – Rojo para representar los resultados.

A.2.3 Casos de Uso

Se identifican los siguientes casos de uso dentro del alcance (ver figura A-3).
Notación: CP significa "caso de prueba".

- Ingreso de CP (sin persistencia).
- Modificación de la configuración de Pruebas.
- Guardar Prueba.
- Baja de CP.
- Modificación de CP.
- Ejecutar Pruebas.
- Listar los objetos verificables.
- Ver detalles de objeto verificable.
- Modificación de los parámetros del objeto verificable.
- Habilitar/ Deshabilitar Pruebas.
- Importar Pruebas.

Diagrama:

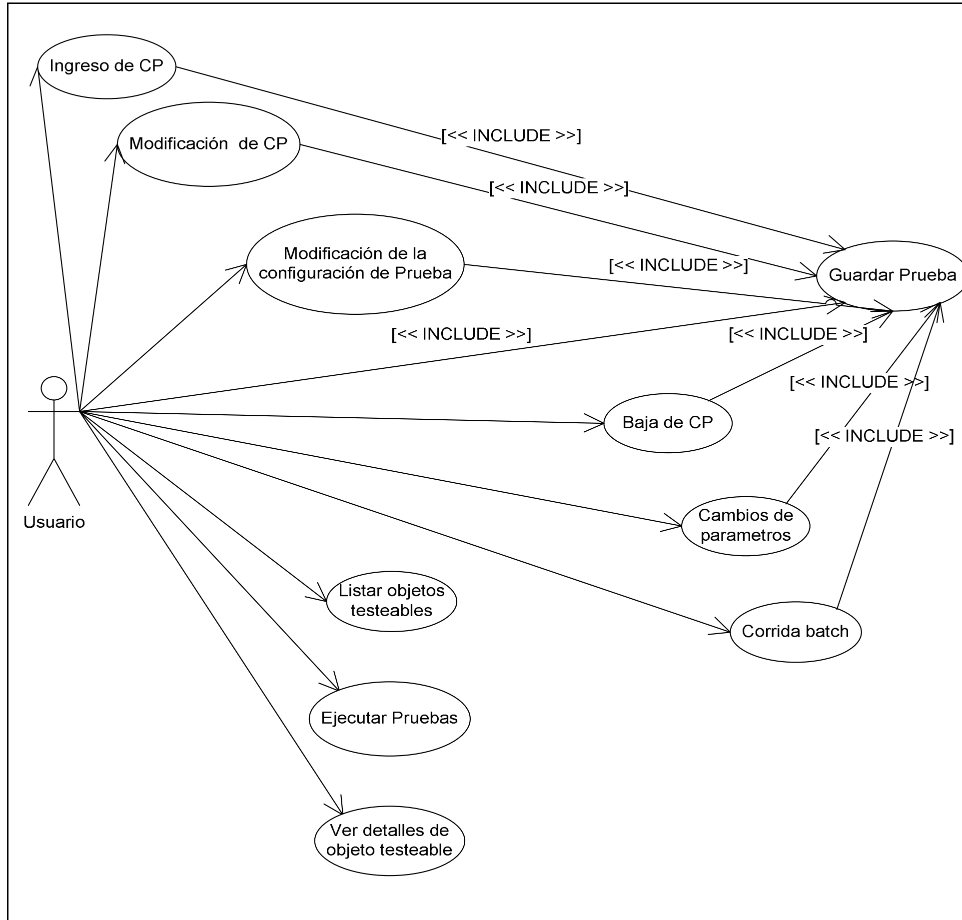


Fig. A- 3: Diagrama de casos de uso (GXUnit1).

Narrativas:

CU1 Ingreso de CP
Se da de alta un nuevo CP. Dado un objeto verificable (procedimiento GX en este primer alcance), se le ingresan los parámetros de entrada y los parámetros de salida esperados en la grilla de ingreso de datos de la pestaña "*test*" asociada al objeto a verificar. Opcionalmente se le ingresa un comentario.

CU2 Modificación de la configuración de Pruebas
Dada una prueba y en presencia de la grilla de ingreso de datos el usuario puede seleccionar o ignorar un subconjunto de los CP asociados a la prueba.

CU3 Habilitar / Deshabilitar Pruebas
En todo momento el usuario puede habilitar o deshabilitar las pruebas. Está opción funcionará como un "check box". Si esta opción está habilitada, se genera el objeto verificador y se guardan los datos de las pruebas asociadas al mismo.

En caso de deshabilitar el check box, al guardar se eliminará el objeto verificador, así como su archivo de casos de prueba.

CU4 Guardar Prueba
Al seleccionar el botón de "guardar", se da de alta en el sistema las pruebas asociadas a un objeto verificable. Se almacenan todos los CP y su información relacionada (entradas, salidas esperadas y de existir, los comentarios), datos: XML y objetos generados (objeto verificador) relacionados al objeto seleccionado así como la selección de corrida por lotes y/o selección de los CP (se guardan todos los CP y no solo los seleccionados).

CU5 Baja de CPs
Se da de baja física el conjunto de CPs seleccionados y su información relacionada: datos XML.

CU6 Modificación de CPs
Se modifican los valores de los parámetros de entrada o salida en la parte "*test*" de los CPs. Independientemente también es posible habilitar o deshabilitar las pruebas para el objeto seleccionado.

CU7 Ejecución de Pruebas
Desde la barra de herramientas del IDE de GX Rocha se selecciona la ejecución de todos los CPs de todas las pruebas asociadas a los objetos verificables previamente habilitadas.

CU8 Listado de objetos verificables
Desde la barra de herramientas de la IDE de GX Rocha se selecciona desplegar el listado de todos los objetos verificables presentes en la KB con la opción "habilitar" encendida.

CU9 Ver detalles de objeto verificable
Desde el listado del objeto se selecciona uno o varios objetos verificables para ver el resultado de su última corrida.

CU10 Modificación de los parámetros del objeto verificable
El usuario modifica los parámetros del objeto verificable, ya sea en su tipo, al agregar un nuevo parámetro, o al quitar un parámetro. El editor se modifica de acuerdo al cambio realizado: se agregan columnas, se quitan columnas o se modifican los tipos.

CU11 Habilitar/Deshabilitar la ejecución por lotes de las Pruebas.
El usuario habilita o deshabilita las ejecuciones de las pruebas a ejecutarse luego del *build*.

CU12 Importar Pruebas
El usuario ingresa los datos para la prueba desde un archivo en formato XML.

Consideraciones sobre el alcance:

Se distinguen 3 tipos de SDTs:

- Simples: SDT de tipo “RECORD” o “STRUCT
- Anidados: SDT A incluye SDT B
- Recursivos: SDT A incluye SDT A

Se abarcan en la solución los SDT de tipo 1 y 2, con atributos de tipo básicos. Se incorporan los tipos de datos “*collection*” de SDT. Los tipos de datos básicos soportados son: *Numeric, Character, Boolean, Date, DateTime, VarChar,* y *LongVarChar*. Las colecciones de tipos de datos simples quedan por fuera del alcance.

A.2.4 Arquitectura

A.2.4.1 Diagrama de Casos de Uso relevantes a la Arquitectura

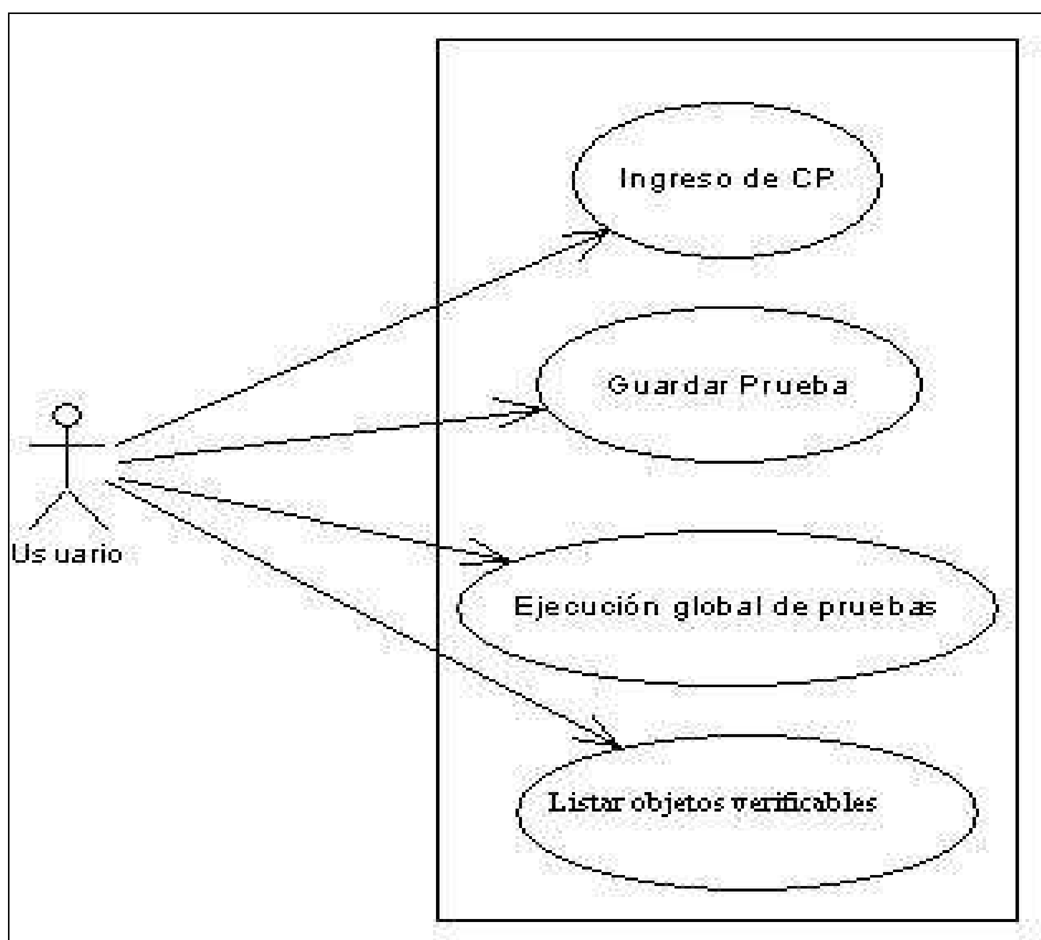


Fig. A- 4: Casos de uso relevantes a la Arquitectura (GXUnit1).

A.2.4.2 Descomposición en SubSistemas

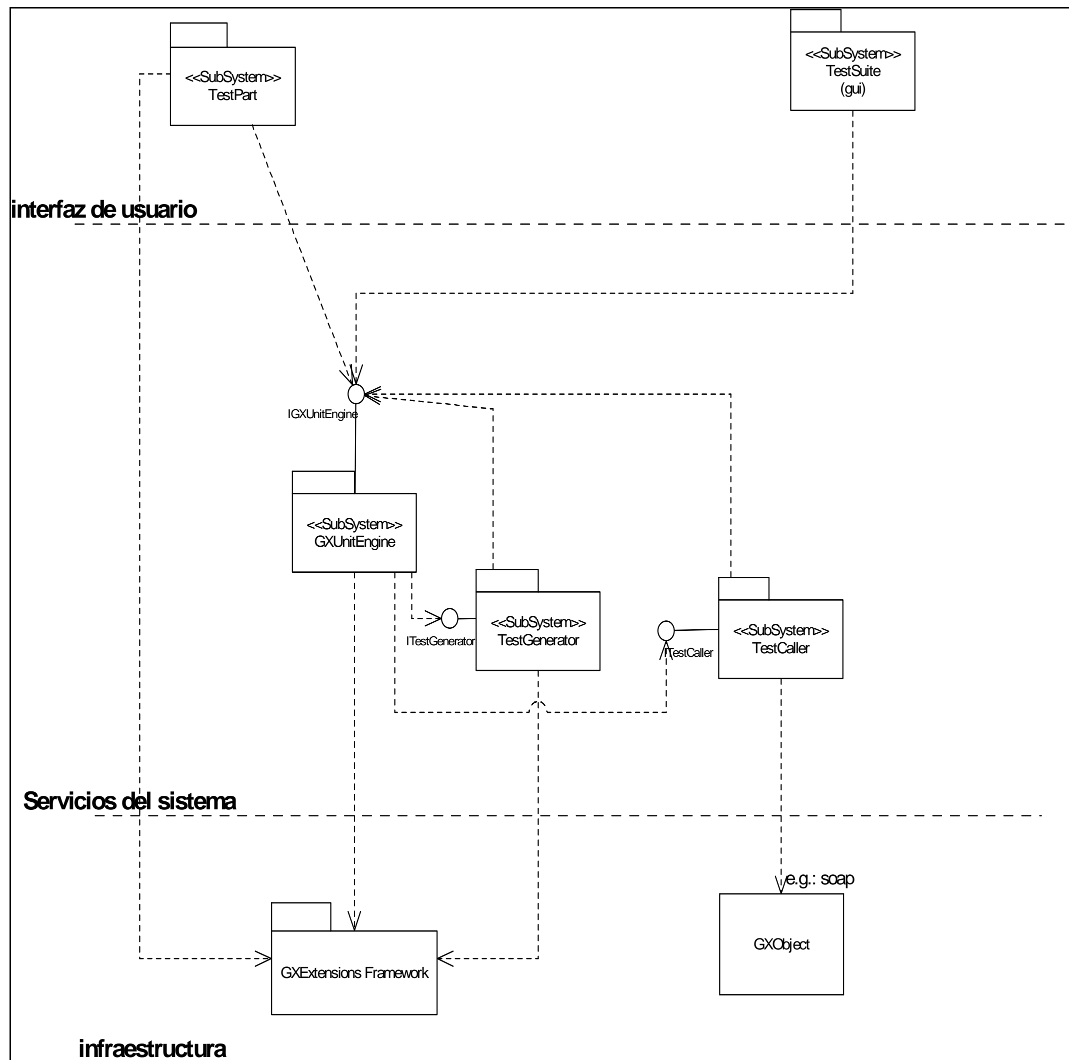


Fig. A- 5: Descomposición en subsistemas (GXUnit1)

Todos los subsistemas a implementar tienen arquitectura de *plug-in*, esto es, se comunicarán con el *Framework* de desarrollo de extensiones de GeneXus. Este ya está implementado y se cuenta con los servicios. A continuación se describe la función básica de cada subsistema:

- **TestPart:** Este subsistema es el encargado de dar de alta casos de prueba para un procedimiento. Será visible en una pestaña al editar un objetos GX procedures. Al persistir el procedimiento, se comunicará con el subsistema GXUnitEngine, para que este se encargue de generar el procedimiento verificador. Este subsistema se concibe para independizar el ingreso de casos de prueba de la generación de los objetos verificadores.

- **TestGenerator:** Este subsistema se encarga de generar la lógica del objeto GX verificador. Se podrá implementar un subsistema de éstos para cada tipo de objeto GeneXus a verificar. Estos subsistemas se registrarán en el subsistema GXUnitEngine. También se encargará de eliminar los objetos generados en caso de que no se quiera probar un objeto al cual anteriormente se le generaron verificadores. La existencia de este subsistema sirve para que la generación de objetos verificadores sea extensible y reusable.
- **GXUnitEngine**: Este subsistema es el encargado de recibir los pedidos de subsistemas del tipo TestPart, y luego generar los objetos pertinentes para probar el objeto en cuestión.
- **TestSuite:** Este subsistema es la interfase mediante la cual los Analistas GeneXus que utilicen la extensión podrán gestionar la ejecución de las pruebas. Desde esta interfase podrán ver que objetos son verificables, seleccionar aquellos que desean probar y solicitar la ejecución de los casos de prueba.
- **TestCaller:** Este subsistema es el encargado de recibir los pedidos para correr objetos de prueba. Se deberá comunicar con la implementación de estos objetos por medio de un protocolo conocido y configurable. Finalmente devolverá los resultados en almacenamiento secundario para ser levantados por TestSuite.

Componentes

- **TestPart.dll:** Depende del componente GXUnitEngine y XMLHelper.
- **GXUnitEngine.dll:** Componente Central del sistema GXUnit.
- **ProcedureTestGenerator.dll:** Depende del componente GXUnitEngine y XMLHelper. Consta del generador de objetos verificadores para procedimientos, y el subsistema que invoca las pruebas generadas.
- **TestSuite.dll:** Depende del componente GXUnitEngine. Implementa la interfase para correr las pruebas.
- **XMLHelper.dll:** Componente que utilizada para la creación de archivos XML y su asistencia.

Interfases

- **IGXUnitEngine**
- **ITestGenerator**
- **ITestCaller**

A.2.4.3 Modelo de Distribución

Diagrama

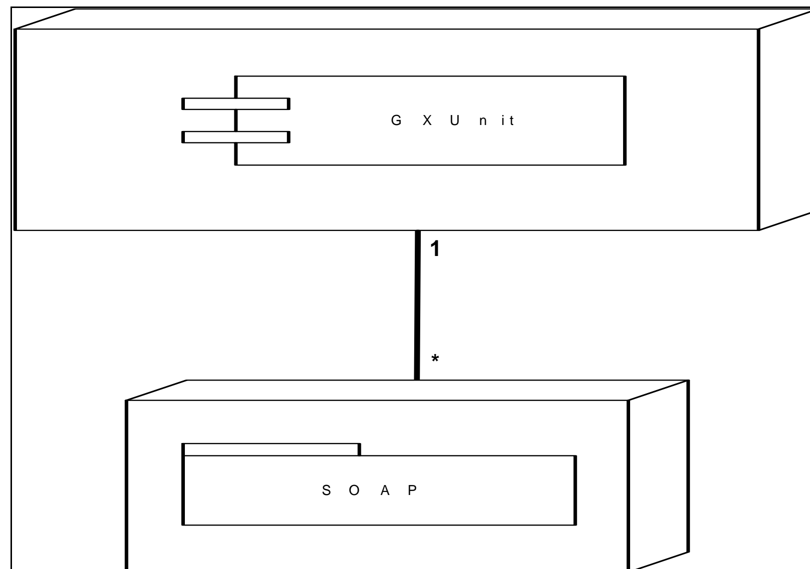


Fig. A- 6: Modelo de distribución (GXUnit1).

Nodos

- **GXUnit:** Este nodo representa la terminal donde el Analista GeneXus ingresará los casos de prueba para los objetos GeneXus a probar y desde donde ejecutará las pruebas.
- **Caller Service:** Este nodo representa los servicios que utilizará el subsistema TestCaller para ejecutar los casos de test.

A.2.5 GXUnit1 en acción

Se desea verificar con GXUnit1 un procedimiento. En las figuras A-7 y A-8 se muestra su regla parm y su código, respectivamente.

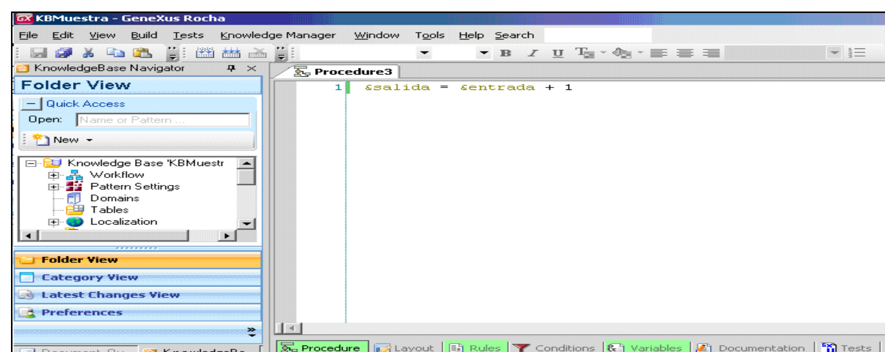


Fig. A- 7: Regla "*parm*"de procedimiento a probar (GXUnit1).

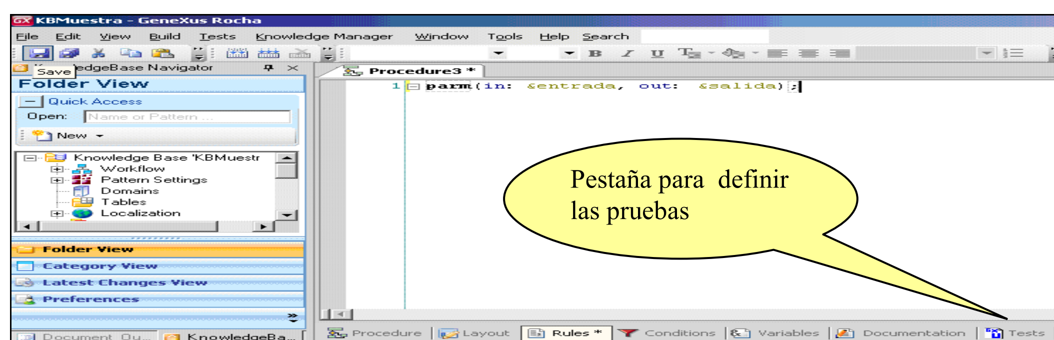


Fig. A- 8: "*Source*"de procedimiento a probar (GXUnit1).

Para introducir los datos de prueba se debe ingresar a la pestaña "*tests*". 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, utilizando la primera columna de la grilla, que casos de prueba se van a correr en la próxima ejecución. También es posible cargar la grilla a partir de un archivo XML. Si se marca "*Generate Tester*" se generará el procedimiento verificador. Ver figura A-9.

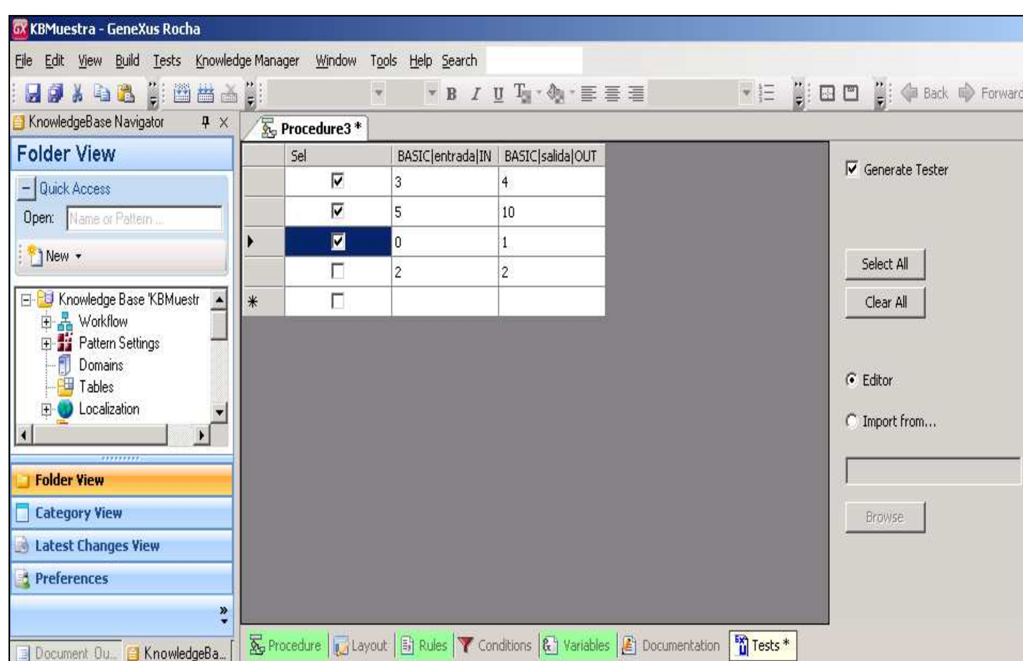


Fig. A- 9: Editor de pruebas GXUnit1.

Al guardar el procedimiento a probar se genera el procedimiento verificador y el archivo XML con los datos de los casos de prueba. El procedimiento verificador se almacena en la carpeta "*Tests*" del modelo y el archivo XML en una carpeta de igual nombre dentro de la carpeta del Modelo. Ver figura A-10.

Fig. A- 10: Creación del procedimiento verificador (GXUnit1).

Se debe ejecutar a continuación un “*Build*”. Ver figura A-11.

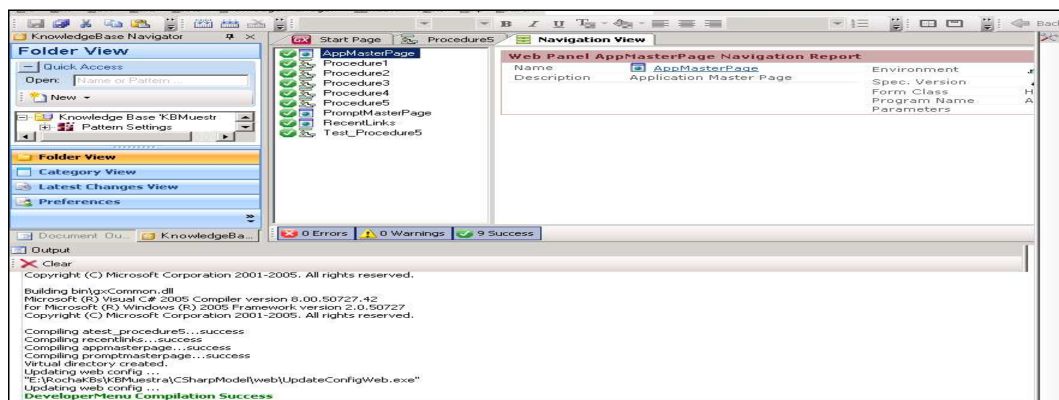


Fig. A- 11: “*Build”* (GXUnit1).

Para ejecutar la prueba se utilizará la opción "*Run Tests*" del menú "*Tests*". Ver figura A-12.

Fig. A- 12: Menú para ejecutar pruebas (GXUnit1).

Posteriormente elegir la opción "*Run*" y habilitar el checkbox en la fila del procedimiento verificador "*TestProcedure*" que se desee ejecutar. (La opción "*Run all Tests after Build*" emula la funcionalidad de proceso por lotes). Ver figura A-13.

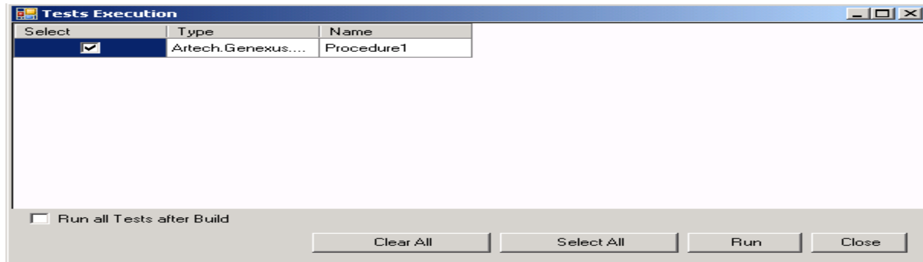


Fig. A- 13: Selección de pruebas a ejecutar (GXUnit1).

Si no salta ninguna excepción en el transcurso de los pasos ejecutados, el resultado de la corrida aparecerá en un archivo XML en la misma carpeta mencionada anteriormente, y en cuyo nombre figura la fecha y hora de corrida. El resultado de la ejecución se podrá observar desde el IDE como se ilustra en figuras A-14 y A-15.

Fig. A- 14: Vista de los resultados de las pruebas -1er..nivel- (GXUnit1).

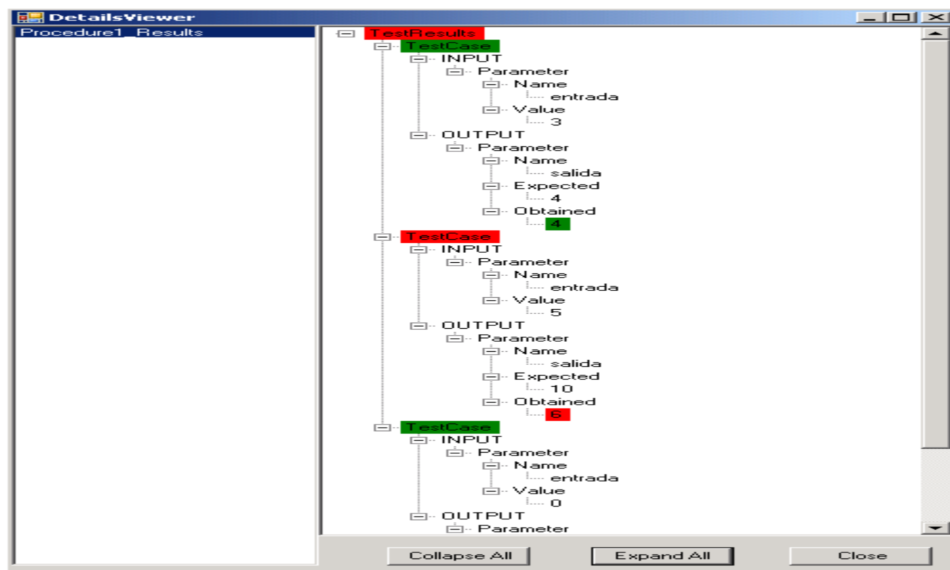


Fig. A- 15: Vista de los resultados de las pruebas –expandida- (GXUnit1).

El funcionamiento de GXUnit1 puede variar un poco si se trabaja con tipos de datos estructurados como parámetros de los procedimientos a probar. En este caso es necesario crear manualmente los archivos XML correspondiente a la prueba. En la tabla A-1 se muestra un ejemplo de los archivos a utilizar para el caso de un procedimiento con dos parámetros SDT, uno de entrada y otro de salida.

SDT - IN	SDT - OUT
<ParametroSDT> <param>**SDT1**</param> <sdt> <Field type="**AtributoComun**"> <param>**Id**</param> <type>**Int32**</type> <value>**43**</value> </Field> <Field type="**AtributoComun**"> <param>**Name**</param> <type>**String**</type> <value>**Juan**</value> </Field> </sdt> /ParametroSDT>	ParametroSDT> <param>**SDT1**</param> <sdt> <Field type="**AtributoComun**"> <param>**Id**</param> <type>**Int32**</type> <value>**44**</value> </Field> <Field type="**AtributoComun**"> <param>**Name**</param> <type>**String**</type> <value>**Juan A.**</value> </Field> </sdt> </ParametroSDT>
SDT – IN(1)	**SDT – OUT(1)**
<ParametroSDT> <param>**SDT1**</param> <sdt> <Field type="**AtributoComun**"> <param>**Id**</param> <type>**Int32**</type> <value>**43**</value> </Field> <Field type="**AtributoComun**"> <param>**Name**</param> <type>**String**</type> <value>**Juan**</value> </Field> </sdt> </ParametroSDT>	ParametroSDT> <param>**SDT1**</param> <sdt> <Field type="**AtributoComun**"> <param>**Id**</param> <type>**Int32**</type> <value>**40**</value> </Field> <Field type="**AtributoComun**"> <param>**Name**</param> <type>**String**</type> <value>**Juan C.**</value> </Field> </sdt> </ParametroSDT>

Tabla A- 1: Archivo XML con datos de prueba SDT (GXUnit1).

En el ingreso de los datos de prueba se debe eligir la ruta donde se encuentran los archivos XML. Ver figura A-16.

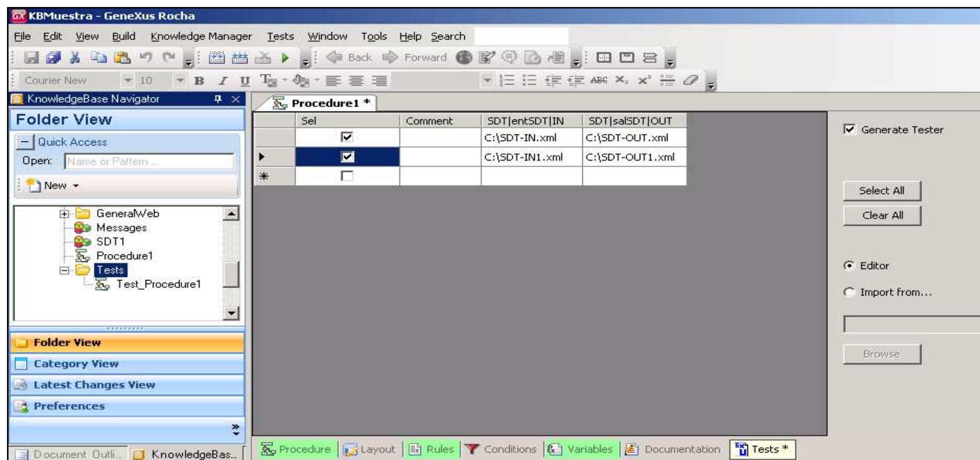


Fig. A- 16: Vista de los resultados de las pruebas –expandida- (GXUnit1).

A.2.6 Grilla del editor de la prueba

El editor de la parte "*tests*", cuya grilla se exhibe en la figura A-17, permite el ingreso de datos para los casos de prueba que se definan para el procedimiento. El editor de la parte se actualiza según nuevos parámetros que se agreguen y/o eliminen a posteriori, conservando los datos anteriores.

Procedure5 *

	Sel	Comment	BASIC|b|IN	BASIC|MyDate|IN	BASIC|b|OUT
▶	☑	test case one	9	11/01/2007	8
*	☐				

Fig. A- 17: Grilla del editor de las pruebas (GXUnit1).

El editor de casos de prueba realiza validación de los datos ingresados, por lo cual, de ingresarse los datos de manera incorrecta, se desplegará un mensaje de error de validación de la celda. Se definen algunas convenciones de formato para determinados tipos, por ejemplo, para Date el formato es "mm/dd/yyyy" y para DateTime el formato es "mm/dd/yyyy HH:MM am/pm".

A.2.7 Comprobación de la base de datos

Esta funcionalidad se entregó como prototipo adicional no integrado a la versión implementada de GXUnit1. Su desarrollo no cumplió con las formalidades de prueba y documentación del resto de la aplicación dado que había sido esta funcionalidad excluida del alcance en las negociaciones con el Cliente.

Características:

- **La especificación** de lo que se desea verificar (valores esperados de tuplas en tablas de la Base de Datos –BD-) se ingresan mediante un archivo xml[83].
- **La obtención de los datos** que se desean verificar se realiza mediante *DataProviders*, a los cuales se les especifica mediante condiciones del tipo *where* las tuplas que son de interés. Estos objetos GeneXus permiten fácilmente obtener datos de la BD en un SDT para luego compararlos con los datos esperados ingresados por el usuario en el XML.
- **La comparación de los datos obtenidos contra los datos esperados** se realiza en el *Procedimiento Verificador* (que es solución del producto final) al cual se le adjunta una sección para la comprobación de los datos obtenidos de la BD.
- **Los resultados de la Prueba** se guardan en el archivo xml resultante del resto de las pruebas. Se le adiciona una parte con el detalle de qué casos fallaron y los datos obtenidos y esperados (el formato es muy similar al de parámetros de tipos de datos simples).
- **La visualización de los resultados de la Prueba** se brinda mediante el visualizador de archivos xml que se provee para el producto final. Estos resultados se visualizan conjuntamente con los resultados del resto de los casos de prueba que no son de la BD.
- **La preparación de la prueba** implica que para cada procedimiento en cuya prueba se desee verificar el estado de la BD se deberá especificar:
  - Tabla(s) que se quieran verificar
  - Condiciones (opcionales) que deban cumplir las tuplas, por ejemplo igualdad, desigualdad, conectivos lógicos. Los registros de interés deben cumplir dichas condiciones.
  - Campos de los cuales se quiera verificar su valor, deben pertenecer a la tabla a verificar.
  - Valor esperado para cada atributo, único. El tipo de datos debe ser compatible con el tipo de datos del atributo.
- **La especificación del estado inicial** se logra con un archivo de sintaxis análoga al ya descripto. Es posible realizar inicializaciones sobre diversas tablas. Para cada tabla, realizar inicializaciones sobre diversos grupos de tuplas. Para cada grupo de tuplas, que se filtra mediante una condición especificada, se asignan una serie de valores a los atributos de la tabla.

Funcionamiento:

En el momento de grabación del procedimiento verificador se generan una serie de objetos de tipo *DataProvider* con el objetivo de leer valores en las tablas luego de ejecutarlo. Se generará un *DataProvider* para cada grupo de tuplas de cada tabla

[83] Su estructura se ofrece en la documentación original y no se reproduce aquí.

sobre la que se desea realizar comprobaciones; o sea, para cada condición sobre tuplas que se especifique en cada tabla en el archivo de entrada, se deberá generar un *DataProvider*. Además se genera un objeto de tipo SDT para realizar la lectura con los distintos *DataProviders*. Cada campo de este SDT se corresponde con los distintos atributos que se desean cargar. Es importante destacar que se utilizará un único SDT con todos los *DataProviders* generados, por lo que el SDT debe tener todos los atributos distintos que se utilizan en dichos *DataProviders*.

Al generar el procedimiento verificador se realizan las inicializaciones deseadas y especificadas en el archivo de inicialización, luego se realiza la llamada al procedimiento a probar y por último se realizan las llamadas a los SDTs para recoger los valores finales de los atributos en las tuplas que se deseaban verificar. Teniendo en cuenta estos valores y los que se especifican en el archivo de valores esperados, se genera el xml en el cual se almacenan los resultados de la corrida.

A.3 GXUnit2

En esta sección se resume la documentación del producto GXUnit2.

A.3.1 Características de la solución técnica

En esta subsección se resumirá acerca de la solución encontrada al problema en cuestión utilizando la interfase de extensiones de GeneXus (de ahora en adelante GX).

Notación: CU significa Caso de Uso
TS significa "TestSet". Nuevo tipo de objeto en la KB creado y mantenido por la extensión. Sus instancias representan pruebas.
CP significa "Caso de Prueba". Consiste en un conjunto de valores de entrada y salidas esperadas para una prueba. Un TS puede contener varios Casos de Prueba.
PVU significa "Procedimiento Verificador de Usuario".

A.3.1.1 Descripción general

GXUnit2 asiste al usuario para realizar verificación unitaria mediante el tipo de objetos ***TestSet***. Un objeto TestSet representa conceptualmente a una suite de casos de prueba que verifican a un único objeto *Procedure*. Tiene (figura A-18) un nombre, una descripción (opcional) y un objeto a verificar asociado de tipo *procedure*. Además, posee un conjunto de casos de prueba y procedimientos verificadores de usuario (PVU).

Fig. A- 18: Objeto TestSet (GXUnit2).

En esta versión de GXUnit solo se soportan objetos *Procedure* cuyos parámetros (en caso de existir) pertenezcan a uno de los siguientes tipos:

- *Numeric*
- *Character*
- *Varchar*
- *Date*
- *LongVarChar*
- *Boolean*

Cada **caso de prueba** (figura A.19) posee un identificador, una descripción, valores para los parámetros de entrada y valores esperados para los parámetros de salida. Los parámetros de entrada y de salida coincidirán con los del objeto a verificar asociado al TestSet. Notar que para aquellos TestSet que verifican un objeto *procedure* que no posee parámetros de entrada ni de salida no se podrán ingresar casos de prueba. En estos casos, la verificación unitaria se realizará únicamente utilizando PVUs (ver siguiente párrafo).

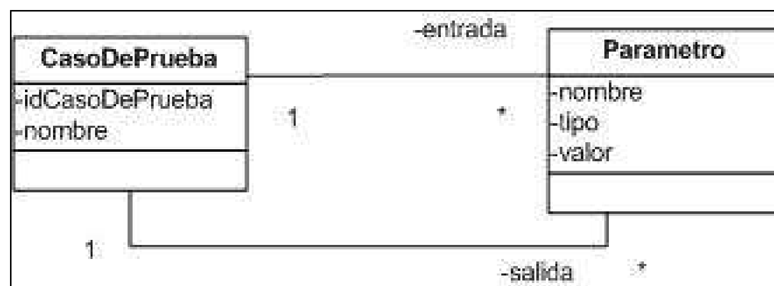


Fig. A- 19: Caso de prueba (GXUnit2).

Un PVU (Figura A-20) es un objeto de tipo procedure creado por el Analista GeneXus para verificar el estado de la base de datos tras la ejecución de un caso de prueba. El usuario deberá programar la lógica de estos procedimientos de forma que se recorran las tablas adecuadas y se verifique que el estado de las mismas es el esperado tras la ejecución de un caso de prueba. Los PVU reciben como entrada los mismos parámetros de entrada del procedimiento que verifican, y un único parámetro de salida de tipo “*booleano*” que debe declararse al final de la regla *Parm* del PVU. Este parámetro de salida indicará si el estado de la base de datos fue el esperado.

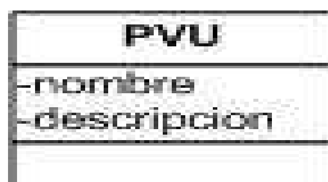


Fig. A- 20: Objeto PVU (GXUnit2).

Se ofrece a continuación un diagrama de los conceptos de GXUnit2:

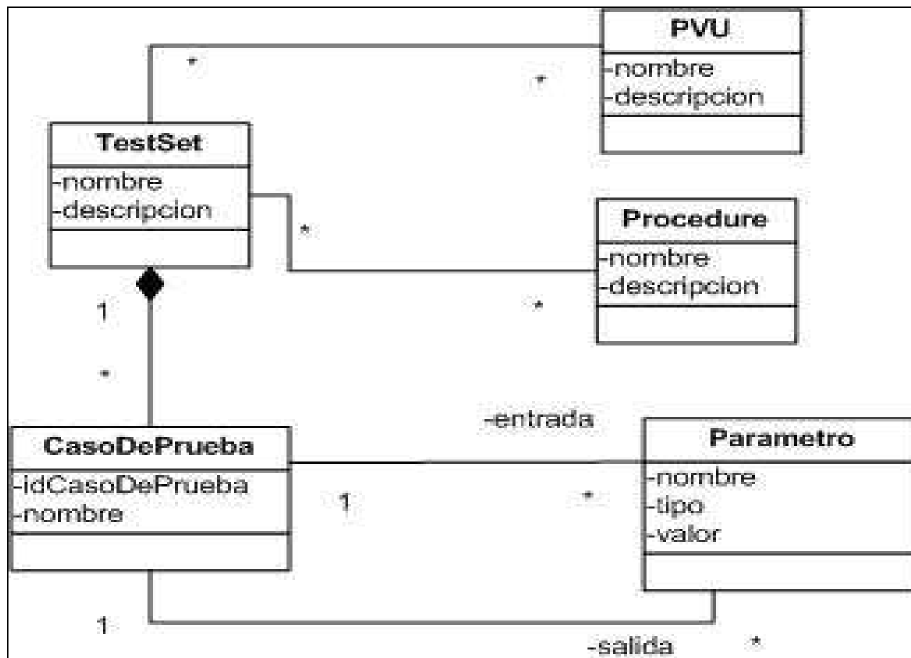


Fig. A- 21: Conceptos (GXUnit2).

A.3.1.2 Decisiones técnicas

Verificación de Objetos Procedimiento GX
La idea utilizada es la generación de un nuevo objeto procedimiento (*procedure*) GX para comandar la verificación.

Invocación
El SDK no provee mecanismos para ejecutar procedimientos existentes en una KB. Por lo que es necesario hacer un procedimiento *main* que invoque al que se quiere probar, para luego utilizarlo desde la extensión. La elección fue respecto al "*Call protocol*" (protocolo de invocación) utilizado por la extensión. Se utilizó el protocolo SOAP, ya que C# provee herramientas sencillas para invocar procedimientos de este tipo además que el protocolo HTTP requiere la programación de las solicitudes y respuestas, en contraposición con el SOAP, que genera el wsdl automáticamente. El procedimiento verificador se consumirá como un *Web Service*.

Invocación de procedimientos con varias variables
El usuario desea tener la posibilidad de probar procedimientos con distinta cantidad de parámetros, así como de distinto tipo. Originalmente se pensó en utilizar un *Call dinámico*, pero GeneXus Rocha no soporta este tipo de invocaciones con cantidad variable de parámetros. Tampoco existe el concepto de "tipo dinámico", por lo que esta solución se descartó. Considerando que esta funcionalidad sería implementada para futura versión de GeneXus Rocha se implementó (de acuerdo con el Cliente) un *Call dinámico* con una cantidad predefinida de variables con tipos conocidos. En este

momento, surgió como requisito de aplicaciones desarrolladas en C# que los procedimientos verificables fueran *main* o bien estuvieran en un código inalcanzable para que sean incorporados al ensamblado (*assembly)*, por lo que requiere generar el procedimiento verificador cada vez que se va a ejecutar (en el caso que deban modificarse éstas referencias). Como resulta necesario volver a generar dicho procedimiento al ejecutar un conjunto de pruebas, es posible en ese momento inspeccionar la KB en busca de procedimientos "verificables" y agregarlos en el código del procedimiento verificador. Se implementó que al momento de ejecutar se genere y compile el *Web Service* con referencias a todos los procedimientos verificables.

Editor de casos de prueba
El ingreso de los casos de prueba se realiza mediante un nuevo tipo de Objeto que se define por parte de la extensión en la KB de GeneXus. Dicho objeto se denominó *TestSet* (TS). Presenta un editor para el ingreso de los datos.

Almacén de los datos de prueba y los resultados
Los datos de prueba se almacenan a través del ambiente GeneXus. Los resultados se almacenan en archivos con formato XML externos a la KB.

Eliminación de procedimientos verificables
Según lo explicado anteriormente se crea un *Web Service* con referencias a todos los procedimientos verificables. Estas referencias son detectadas por GeneXus como *hard*, por lo cual no permite la eliminación de ningún procedimiento verificable una vez que existe el *Web Service*. Esto se puede evitar, transformando las referencias *hard* a *weak*. Dado que esta solución se publicó sobre el final del desarrollo se convino con el Cliente no implementarla, dejándola debidamente documentada. En tanto, se permite eliminar procedimientos verificables haciendo "clic derecho" sobre ellos en la *toolwindow* (de GX) y eligiendo la opción "*Delete procedure*". Si algún objeto tiene una referencia *hard* hacia el (y no es un objeto verificador) no permite la eliminación. De lo contrario se eliminan el *Web Service*, los objetos de prueba que lo referencian y el procedimiento.

Identificación de procedimientos verificadores de usuario (PVU)
Al seleccionar PVUs para un TS, el usuario sólo tendrá disponibles aquellos que son potencialmente PVU. El problema de detectar cuales procedimientos son "potencialmente PVUs[84]" del que se está probando, se resolvió mostrando todos aquellos procedimientos que de acuerdo a sus parámetros sean potencialmente PVUs. Se descartó la opción de indicarlo como una parte donde se informara si es PVU además de la lista de procedimientos que prueba, dado que GeneXus Rocha no implementa el concepto de "generalización" por lo que la parte debería agregarse al tipo de objeto procedimiento. Esto haría que todos los procedimientos de una KB donde está instalada la extensión tengan esta parte. Además, hay muchas extensiones

[84] Se dice que un procedimiento A es "potencialmente PVU" de otro B cuando la única salida de A es de tipo *"Booleana"* y todos los parámetros que recibe son del mismo tipo que los de B.

agregando partes a los objetos, por lo que se puede afectar la facilidad de uso del producto con varias extensiones instaladas.

Invocación del *Web Service* cuando su compilación ha fallado
Desde la extensión se puede iniciar la compilación de un objeto GeneXus (es lo que hace GXUnit2 al modificar el *Web Service*), pero no es posible detectar si la compilación fue exitosa o no. Es por esto que la extensión consume el *Web Service* independientemente del resultado de su compilación, lo cual trae como consecuencia que se consuma el último *Web Service* que ha sido correctamente compilado. Si no existe, se informará que no se pudieron ejecutar los casos de prueba. El SDK permitirá en el futuro verificar si la compilación fue exitosa.

Modificación del objeto verificable
Al ejecutar un TS se verifica si el *Web Service* debería ser generado nuevamente. Pero si algún procedimiento verificable ha tenido modificaciones que no incluyan a la regla *parm*, el *Web Service* no requerirá cambios, por lo que no se compilará y se usará la última versión compilada del procedimiento modificado. Dado que desde la extensión no se puede verificar si un objeto necesita volver a ser compilado sin ejecutar un *build* se decidió, por sugerencia del Cliente, implementar que el usuario pueda forzar una compilación al ejecutar un TS.

Diseño de Subsistemas
Dadas las características del proyecto y por aspectos puramente técnicos y de diseño de GeneXus, en especial de la interfase que provee para desarrollar extensiones (SDK), existen implicancias en algunas decisiones que limitan el diseño de GXUnit. Por un lado, no es posible una separación en capas "Presentación" y "Lógica" ya que GeneXus impone dependencias que rompen esta posible separación. Además por estas dependencias y la poca cantidad de clases, no existe una gran modularización a realizar dentro del diseño de los subsistemas.

A.3.2 Casos de Uso

Se identifican los siguientes casos de uso dentro del alcance.

- Crear TS.
- Eliminar TS.
- Agregar CP a TS.
- Seleccionar nivel para "Logs"
- Crear PVU.
- Eliminar PVU.
- Asociar PVU a TS.
- Modificar TS.
- Ejecutar TS.
- Modificar un CP de TS.
- Ver histórico de Ejecuciones.
- Eliminar CP de TS.
- Modificar PVU
- Desasociar PVU de TS.

Diagrama:

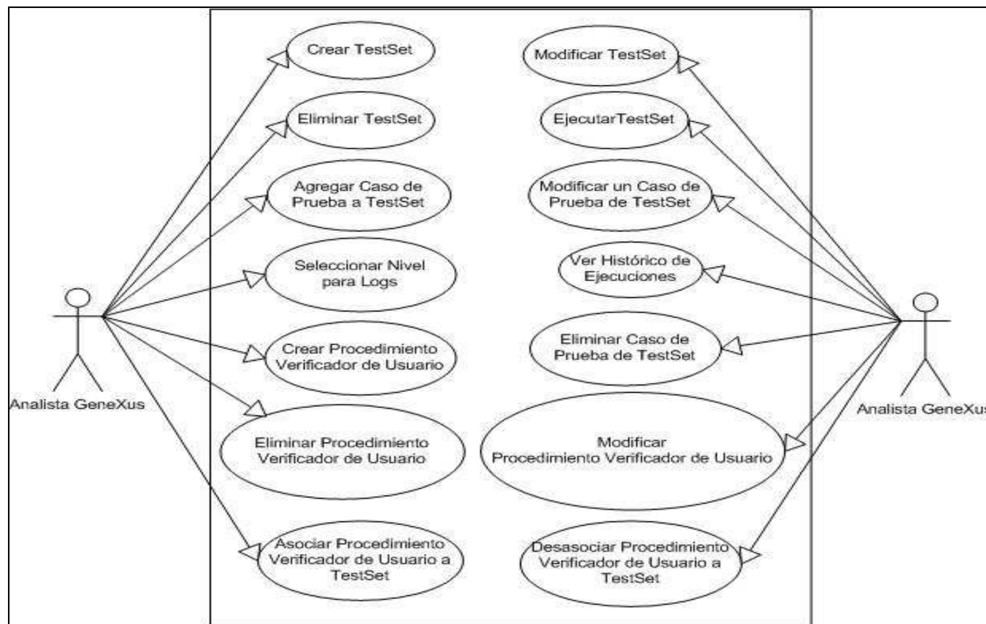


Fig. A- 22: Diagrama de casos de uso (GXUnit2).

Narrativas

CU1 Crear TestSet (TS)
Este caso de uso permite al Analista GeneXus crear un objeto "TS". El Analista crea el objeto y elige un procedimiento verificable a probar. Nota: El objeto TS se asocia al procedimiento verificable.

CU2 Modificar TS
Este caso de uso permite al Analista GeneXus modificar la descripción así como también habilitar o deshabilitar la ejecución de todos los Procedimientos Verificadores de Usuario (PVU) asociados al objeto "TS" para cada Caso de Prueba que este contenga.

CU3 Eliminar TS
Este caso de uso permite al Analista GeneXus eliminar un objeto "TS" existente en el sistema.

CU4 Crear Procedimiento Verificador de Usuario (PVU)
Este caso de uso permite que el Analista GeneXus cree un PVU que efectuará validaciones tras la ejecución de un caso de prueba. Será condición del sistema que el procedimiento creado tenga un solo parámetro de salida del tipo "*booleano*". Sus parámetros de entrada deben coincidir en cantidad, tipo y dimensión con los del procedimiento a probar.

CU5 Modificar un PVU
Este caso de uso permite que el Analista GeneXus modifique algunas de las propiedades de un PVU. Entre otras modificaciones el usuario puede cambiar el código a ejecutar por dicho procedimiento.

CU6 Eliminar un PVU
Este caso de uso permite al Analista GeneXus eliminar un PVU.

CU7 Agregar Caso de Prueba a un objeto TS
Este caso de uso permite al Analista GeneXus agregar un Caso de Prueba a un objeto TS ya existente. El Analista especifica los valores de entrada y las salidas esperadas (estas últimas podrán ser un subconjunto de todas las que tiene el Procedimiento). Nota: Se contemplará el ingreso de parámetros de tipo SDT, el cual es diferente al de parámetros básicos.

CU8 Modificar un Caso de Prueba de un objeto TS
Este caso de uso permite al Analista GeneXus modificar el nombre, descripción, datos de entrada o resultados esperados de un caso de prueba.

CU9 Eliminar un Caso de Prueba de un objeto TS
Este caso de uso permite al Analista GeneXus eliminar un caso de prueba de un TS particular.

CU10 Asociar un PVU a un objeto TS
Este caso de uso permite que el Analista GeneXus asocie un procedimiento verificador de usuario.

CU11 Desasociar un PVU de un objeto TS
Este caso de uso permite que el Analista GeneXus elimine una asociación existente entre un Procedimiento Verificador de Usuario y un TS.

CU12 Ejecutar objetos TSs
Este caso de uso permite que el Analista GeneXus ejecute uno o más TSs existentes en la KB actual, mostrando un informe del resultado de la ejecución.

CU13 Selección del nivel para bitácoras (logs)
Este caso de uso permite que el Analista GeneXus seleccione el nivel de detalle que tendrán logs generados por las ejecuciones de los TSs.

CU14 Ver histórico de ejecuciones
Este caso de uso permite que el Analista GeneXus consulte los logs generados en cada ejecución de un TS. Cabe aclarar que el nivel de detalle de la información que guarda el sistema de una ejecución (log), depende del nivel seleccionado por el usuario a través del caso de uso CU13. El Sistema deberá brindar varios criterios de búsqueda para que el usuario pueda visualizar la corrida que le interese.

<u>Consideraciones sobre el alcance:</u>

Reacción ante cambios en los procedimientos a verificar
Al modificar los parámetros de un procedimiento a probar, será necesario que esto se refleje en el TS, así también como al eliminar procedimientos verificadores de usuario. Se deberá implementar una forma gráfica que le informe al usuario, qué fue lo que cambió y si los valores antes ingresados para los casos de prueba de un determinado TS tienen utilidad, mostrarlos.

Ejecución de TS
Se excluyen del alcance los SDT. Se invocará mediante *Web Service* al procedimiento que ejecute las pruebas.

Ejecución por Lotes
Debido a que el servicio que se utiliza para compilar el *Web Service* se encuentra accesible solamente desde el IDE de GeneXus, es técnicamente imposible implementar este requerimiento hoy en día. Este servicio obliga a que la ejecución de TS se realice siempre mediante la IDE, excluyendo la forma de ejecución por lotes.

A.3.3 Modelo de Dominio

<u>Diagrama</u>

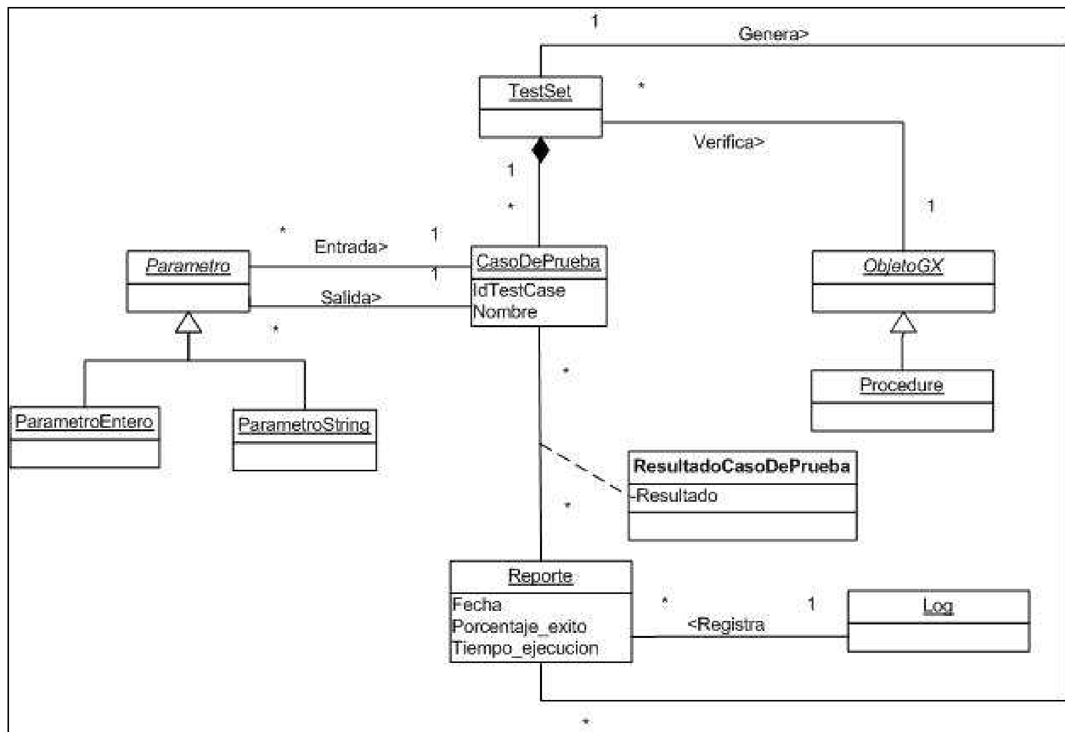


Fig. A- 23: Modelo de Dominio (GXUnit2).

Observaciones:

- **La entidad *Parametro*** representa los parámetros que recibe como entrada y espera como salida un caso de prueba. Es una generalización de los distintos tipos de parámetros que provee GeneXus. ***ParametroEntero*** y ***ParametroString*** son dos tipos de parámetros que pueden recibir/esperar los casos de prueba
- **La entidad *ObjetoGenexus*** es una generalización de todos los objetos provistos por GeneXus. En particular, de estos objetos nos interesa el objeto ***Procedure*** que será el primero en ser verificable.
- **La entidad *CasoDePrueba*** posee un conjunto específico de parámetros de entrada y parámetros de salida. Además, debe poder ser identificado de forma única por el sistema.
- ***TestSet*** se compone de uno o más ***CasoDePrueba***. El ***TestSet*** verificará un único ***ObjetoGX***.
- ***Reporte:*** Las sucesivas ejecuciones de un ***TestSet*** generarán varios ***Reporte***. Cada uno de estos últimos contendrá la fecha de la ejecución, el porcentaje de casos de prueba exitosos y el tiempo de ejecución de todas las pruebas. La información de todos los reportes generados se registrará en un ***Log***.

A.3.4 Arquitectura

La arquitectura del sistema quedó determinada en particular por el caso de uso "Ejecutar TestSet". Aquí una de las decisiones diseño fue la elección de un *Web Service* para la invocación de procedimientos de la KB a probar. Dicha decisión fue lo que mayormente determinó la arquitectura. Algunas de las ventajas y desventajas que se encontraron de la utilización de un *Web Service* fueron las siguientes:

Ventajas:

- Simple generación desde GeneXus Rocha.
- Simple utilización desde .NET
- Estándar bien definido, lo que permite que en el futuro otras aplicaciones. realicen pruebas sobre procedimientos de la KB.

Desventajas:

- Requiere trabajar en ambiente web, ya que es necesario trabajar con un servidor web para exponer el *Web Service*.

A.3.4.1 Diagrama de Casos de Uso relevantes a la Arquitectura

Se incluye en el diagrama de la figura A-24 la trazabilidad del modelo de casos de uso al modelo de diseño.

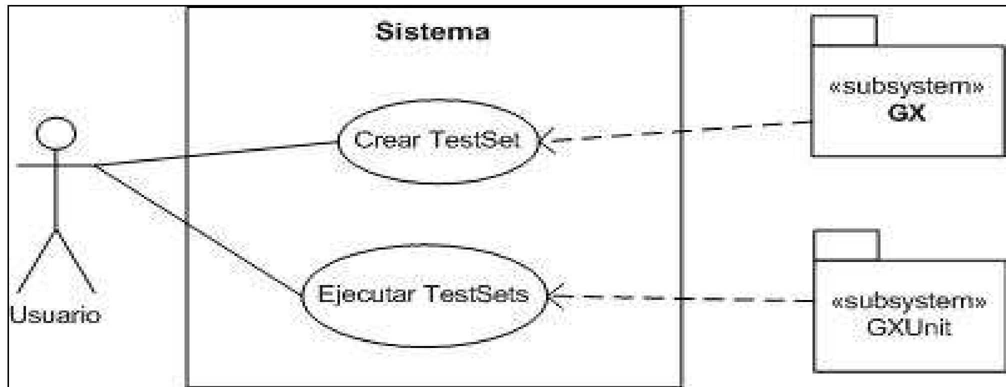


Fig. A- 24: Casos de uso relevantes a la Arquitectura (GXUnit2).

A.3.4.2 Descomposición en subsistemas

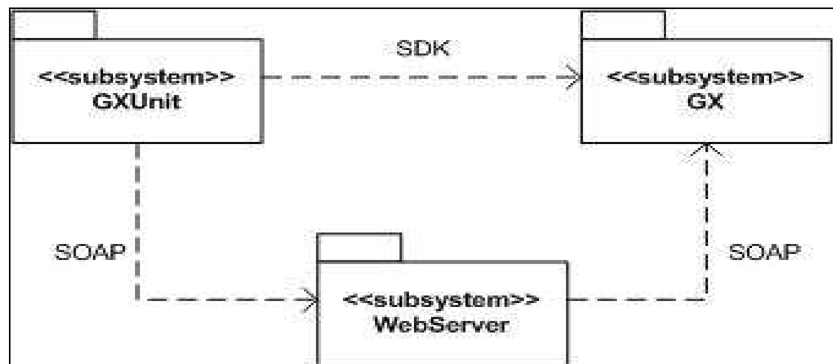


Fig. A- 25: Descomposición en subsistemas (GXUnit2).

Subsistema GXUnit: El objetivo de este subsistema es proveer la interfase para el usuario con la extensión y además encapsular la lógica de la comunicación con el ambiente GeneXus. El subsistema GXUnit se separa en 2 módulos, uno correspondiente a todo lo referente al TestSet (TestSetComponent) y otro (GXComponent) para la lógica de uso y comunicación con el subsistema GX (ambiente GeneXus). Esto se muestra en la figura A-26.

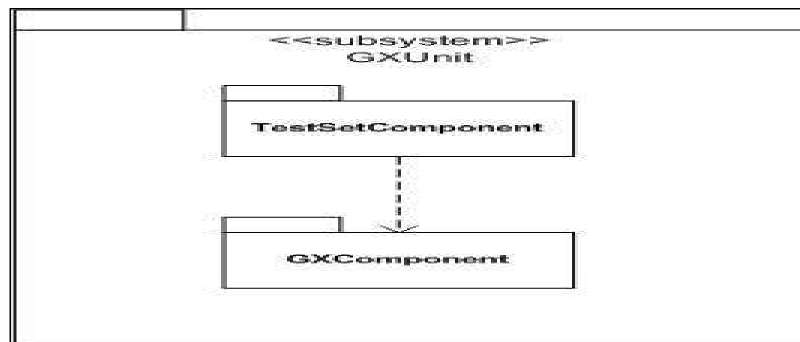


Fig. A- 26: Subsistema GXUnit (GXUnit2).

Subsistema GX: Este subsistema es el que dará la interfase que proveerá de todos los servicios necesarios para la interacción con el ambiente GeneXus. Este

subsistema fue agregado para la mayor comprensión del planteo de la arquitectura, pero el mismo ya está implementado, al ser el ambiente GeneXus desarrollado por Artech.

Este subsistema provee dos tipos de interfases bien diferenciadas. Por un lado ofrece la interfase mediante la API de la SDK la cual es extensa y brinda todos los servicios para comunicarse con el ambiente GeneXus Rocha, por ejemplo la interacción con los objetos de la KB, manejo de menú personalizado, agregar funcionalidades, tipos de objetos nuevos. Por otra parte, se le agrega una interfase mediante la publicación de un *Web Service* la cual permite la invocación de objetos GeneXus *procedure* pertenecientes a una KB de GeneXus Rocha. Podría representarse de la siguiente manera (figura A-27):

Fig. A- 27: Interfase IwebService (GXUnit2).

El *string request* corresponde a un xml enviado mediante el *Web Service*, solicitando la ejecución de un procedimiento en particular, mientras que el string retornado por la función es el xml respuesta del *Web Service*. El uso de xml en esta comunicación es razonable ya que la comunicación con el *Web Service* es por medio de mensajes SOAP mediante el protocolo HTTP.

A.3.4.3 Diagrama de clases

En la figura A-28 se ofrece el diagrama de clases. A continuación se resume acerca de las clases principales.

- **TestSetPart:** Almacena toda la información de un TS, como son el procedimiento asociado, el conjunto de CPs y su conjunto de PVUs.
- **TestSetEditor:** Implementa la interfase a presentar para edición del TS al Analista GeneXus. La misma es editor que permite al Analista GeneXus ingresar toda la información al TS.
- **TestCase:** Almacenar los datos de un caso de prueba ingresado por el usuario. La clase tiene como atributos, el nombre del caso de prueba y una lista de parámetros, clase que almacena los datos: *name* (nombre del parámetro), *type* (tipo del parámetro), *mode (in, out, in/out)*, *value* (valor ingresado).
- **UPT:** Almacenar los datos de un PVU asociado a un TS. Esta clase tiene como atributos, el "id" del procedure asociado y una descripción del mismo.
- **GXComponentController:** Implementa todas las operaciones provistas por la interfase **IGXComponent**. Es la encargada de realizar la comunicación de objetos correspondiente para cada operación y por tanto implementar la funcionalidad debida.

- **XMLComponent:** Implementa todo el manejo de xml en el sistema. Generación de xml para la funcionalidad del *log*, así como también para los xml generados para la comunicación con el subsistema GX mediante el *Web Service*.
- **LogComponent:** Provee todas las funcionalidades para el manejo de *log*s.
- **WebServiceManager:** Manejador de los servicios que provee el *Web Service* para la ejecución de los casos de prueba.
- **SDK:** Representa de forma simbólica la interfase que provee GeneXus Rocha mediante la API de la SDK.

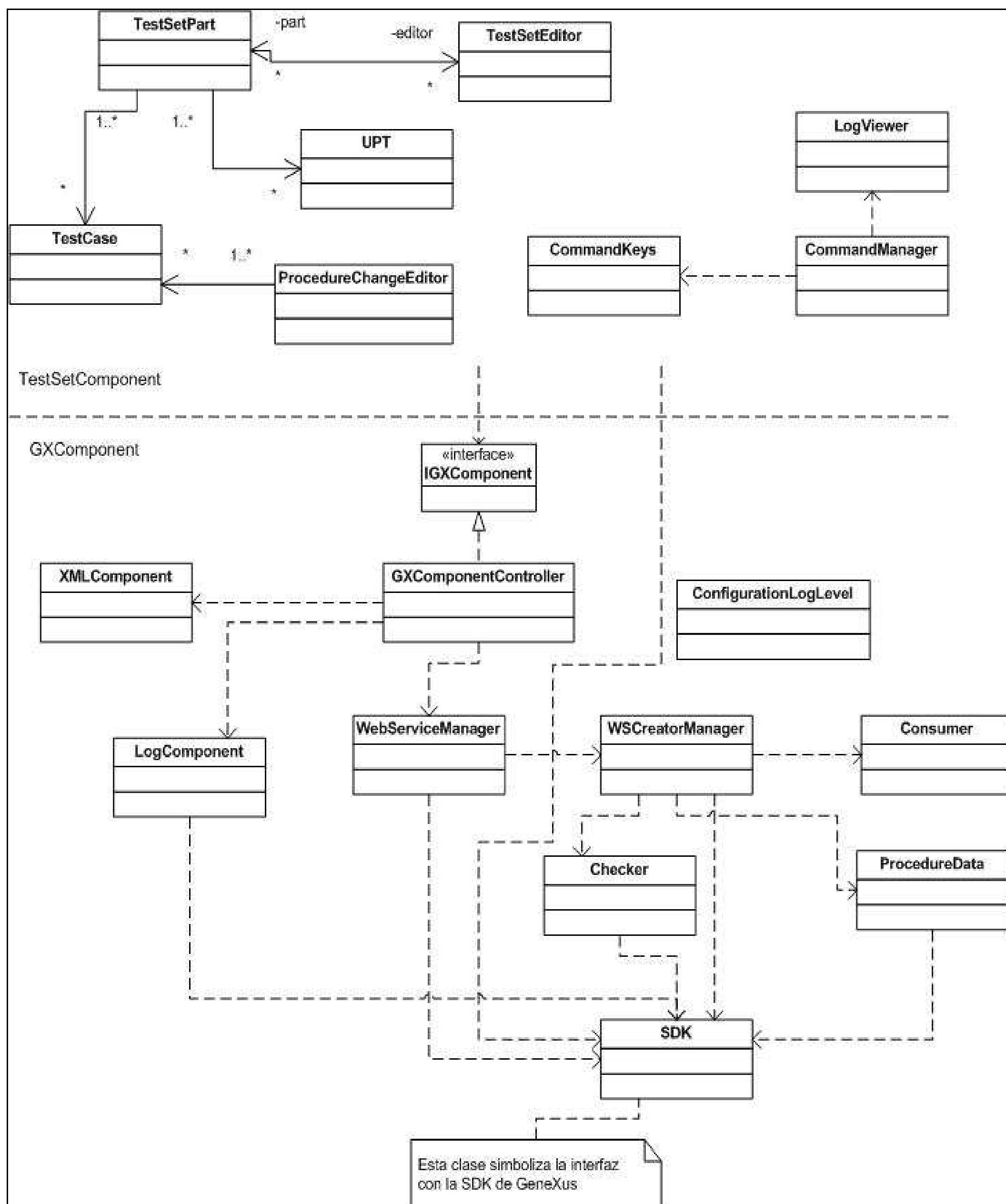


Fig. A- 28: Diagrama de clases (GXUnit2).

Interfase

Interfase IGXComponent: Provista por el módulo GXComponent para la mayor abstracción posible sobre las operaciones que involucran la comunicación con el subsistema GX.

A.3.5 GXUnit2 en acción

El objeto TestSet se crea a partir de la pantalla habitual de creación de objetos de GeneXus, tal como se muestra en la figura A-29 donde se crea un TestSet de nombre TestSet1.

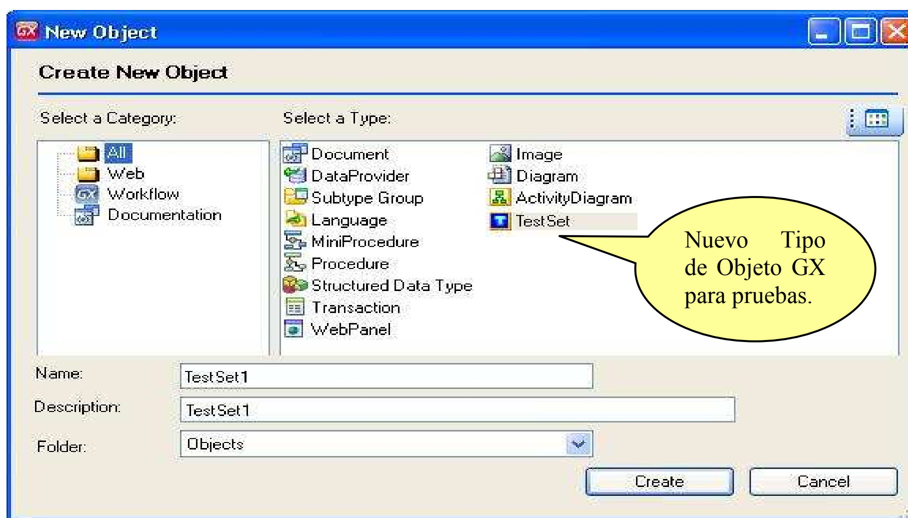


Fig. A- 29: Creación de objeto TestSet (GXUnit2).

Luego de creado un TestSet se presenta una pantalla para definir la prueba. Para esto se solicitará el nombre del procedimiento verificable a probar, los casos de prueba y los PVUs. Ver figura A-30, donde se muestra la pantalla para definir la prueba en un TestSet denominado TestSetSumador.

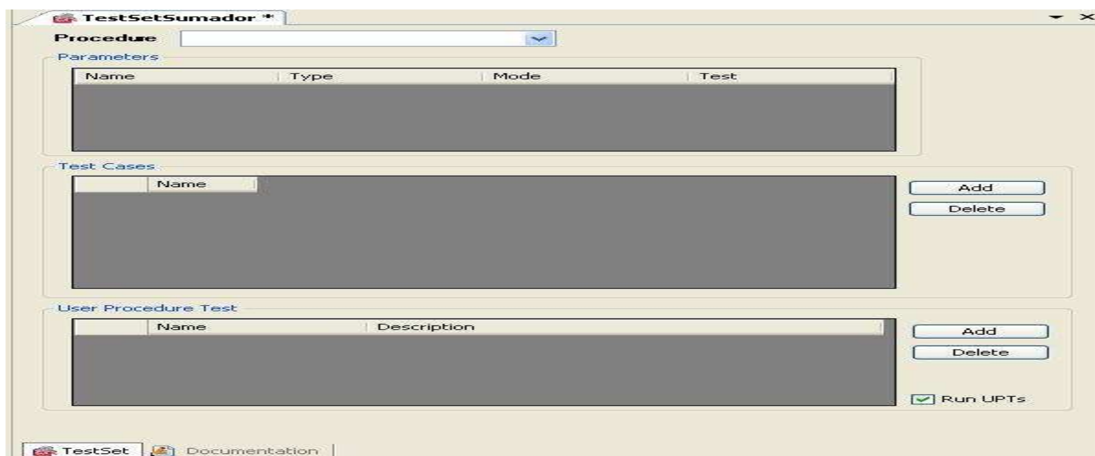


Fig. A- 30: Editor del objeto TestSet (GXUnit2).

Con la instancia TestSetSumador se desea probar un procedimiento verificable existente en la KB llamado “sumador”, que recibe como entradas dos parámetros (enteros, de largo 4, sin decimales) y ofrece como salida otro parámetro (de igual tipo y dimensión) con la suma de los valores de los parámetros de entrada. La regla parm del procedimiento sumador es parm(in:entrada1, in:entrada2, out:salida). El combo de selección de procedimientos permite ubicarlo y seleccionarlo. El editor despliega los parámetros del procedimiento en una grilla tal como se observa en la figura A-31.

Procedure: Sumador

Parameters

Name	Type	Mode	Test
entrada1	NUMERIC(4.0)	IN	☑
entrada2	NUMERIC(4.0)	IN	☑
salida	NUMERIC(4.0)	OUT	☑

Fig. A- 31: Procedimiento verificable a probar y sus parámetros (GXUnit2).

El Analista puede indicar cuales de estos parámetros (de salida) interesan para la prueba. Para efectuar dicha selección los marca con un tilde haciendo clic en la columna rotulada “Test” para cada parámetro a seleccionar.

Una vez establecido esto, se presentará una grilla donde se podrán ingresar, sucesivamente, todos los casos de prueba. Las columnas de la grilla corresponderán a cada uno de los parámetros involucrados y cada fila corresponderá a un nuevo caso de prueba. Se hace control de tipos. El procedimiento a probar se ejecutará una vez por cada fila de la grilla cuando se dispare la prueba. Ver figura A-32.

Procedure: Sumador

Parameters

Name	Type	Mode	Test
entrada1	NUMERIC(4.0)	IN	☑
entrada2	NUMERIC(4.0)	IN	☑
salida	NUMERIC(4.0)	OUT	☑

Test Cases

Name	entrada1	entrada2	salida
Caso1	2	2	4
Caso2	1	2	4

Add Delete

User Procedure Test

Name	Description

Add Delete

Run UPTs

Fig. A- 32: Editor de TestSet y su grilla para ingreso de casos de prueba (GXUnit2).

Finalmente es posible indicar uno o más PVU para verificaciones adicionales que el Analista estime pertinente realizar (control de contenido de la base de datos). Se propone al Analista como candidatos, a seleccionar desde un combo, los procedimientos que tienen igual número y tipo de parámetros que el procedimiento a verificar y uno adicional, como salida, de tipo "*booleano*". Es posible indicar si se deben ejecutar o no los PVU seleccionados.

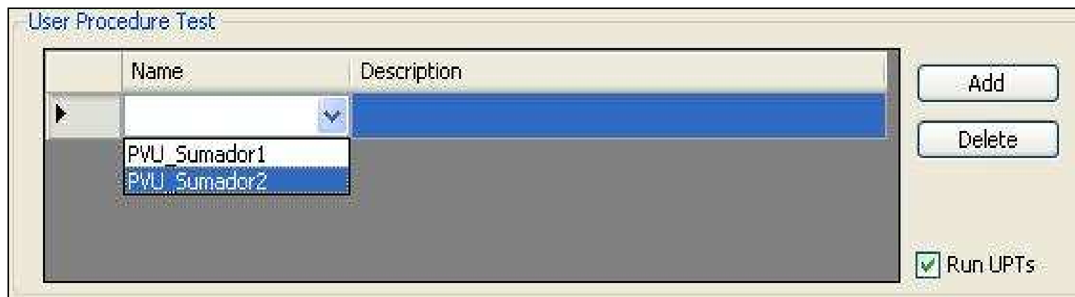


Fig. A- 33: Combo de PVUs en el editor del TestSet (GXUnit2).

Para ejecutar las pruebas se utiliza el menú de la *toolwindows*, al que se le agregó la opción "*Testeable Procedures*" (ver figura A-34).

Fig. A- 34: Menú para ejecución de pruebas (GXUnit2).

La ventana que se abre una vez seleccionada dicha opción muestra todos los procedimientos verificables y sus TestSet asociados, en una estructura arborescente. En la figura A-35 se presenta dicha pantalla para el procedimiento "sumador" y el TestSet que lo prueba, "TestSetSumador".

Fig. A- 35: Ventana para lanzar ejecución de pruebas (GXUnit2).

Generalizando, la ventana "*Testeable Procedures*" luciría como se muestra en la figura A-36 (si se asume la existencia de tres procedimientos verificables con uno o varios TestSet asociados). Como se muestra en dicha figura, el Analista puede, con un clic, seleccionar o deseleccionar tanto los procedimientos a verificar como los conjuntos de casos de prueba (TestSet) a ejecutar.

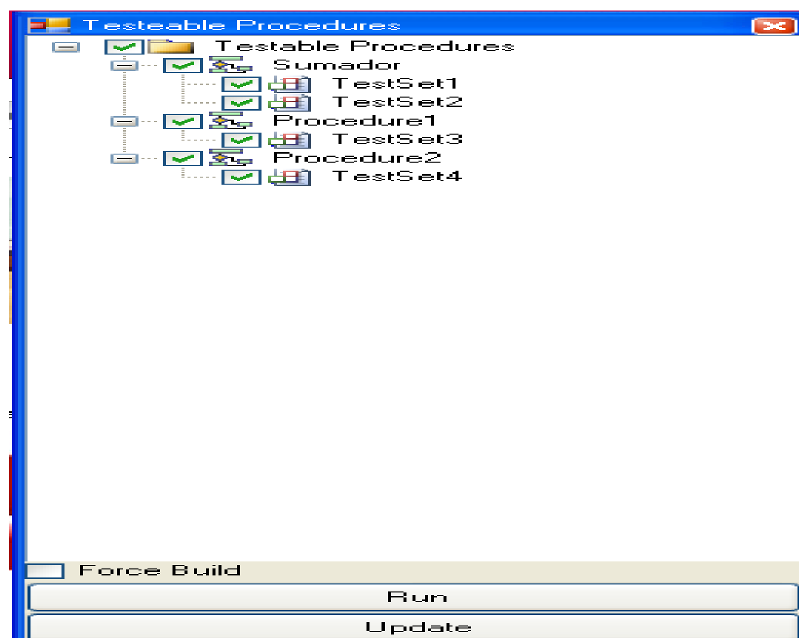


Fig. A- 36: Otra vista de la ventana para lanzar pruebas (GXUnit2).

Luego de solicitar la ejecución con el botón "Run", para el caso de la figura A-35, se ejecutan las pruebas y se podrá observar una pantalla como la que se brinda, a modo de ejemplo, en la figura A-37.

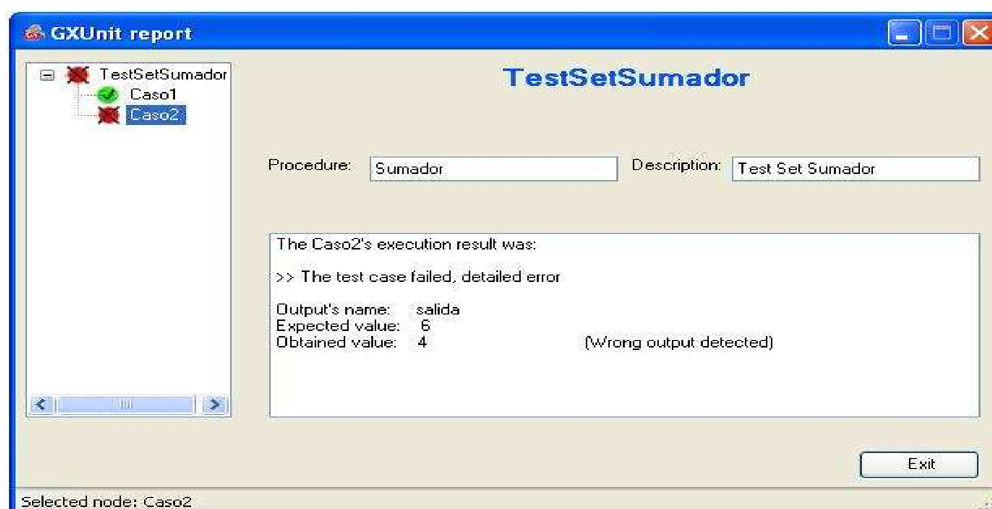


Fig. A- 37: Reporte de ejecución (GXUnit2).

En el panel izquierdo del reporte de ejecución de la figura A-37 se puede ver un listado de los TestSet ejecutados con íconos que indican si el resultado obtenido fue el esperado para todos los casos de prueba (ícono verde) o si fallaron uno o más casos de prueba del mismo (ícono rojo). Esta información se encuentra también detallada en el panel derecho de la ventana del reporte. Debe recordarse que un caso de prueba puede fallar tanto si uno de los parámetros de salida esperados (y marcados para revisar en TestSet) no contiene el valor esperado, si al menos un PVU asociado devuelve ***false*** o si el procedimiento a verificar no pudo ser ejecutado[85]. Al abrir el nodo del TestSet en el árbol, se puede apreciar cuales casos de prueba fueron exitosos y cuales fallaron. Si se selecciona un caso de prueba en particular, se puede ver más detalles con respecto al caso de prueba específico.

Además de presentar el reporte de resultados, la extensión registrará la ejecución de TestSets. El resultado será la creación/modificación de un archivo XML. Más adelante se resumirá acerca de la forma de configurar el nivel de detalle con el cual se graba el log y la ruta donde se ubica (este log no se guarda en la KB). Si el Analista desea visualizar dicho histórico de ejecuciones cuenta con un ítem en el menú del IDE dedicado a GxUnit (ver figura A-38).

Fig. A- 38: Menú GXUnit (GXUnit2).

La pantalla del visor de logs se muestra en las figuras A-39 y A-40. Para conocer el nombre y el tipo de los parámetros de entrada o salida de un caso de prueba, bastará posicionar el puntero del ratón sobre el nombre de la columna

[85] Esto puede deberse a cambios en el procedimiento a verificar tras la creación del TestSet..

correspondiente y se desplegará la información deseada. Haciendo clic sobre el título de una columna en las grillas de información de los PVUs y de resultados de ejecución, se ordenarán las filas de la grilla según el valor de la columna.

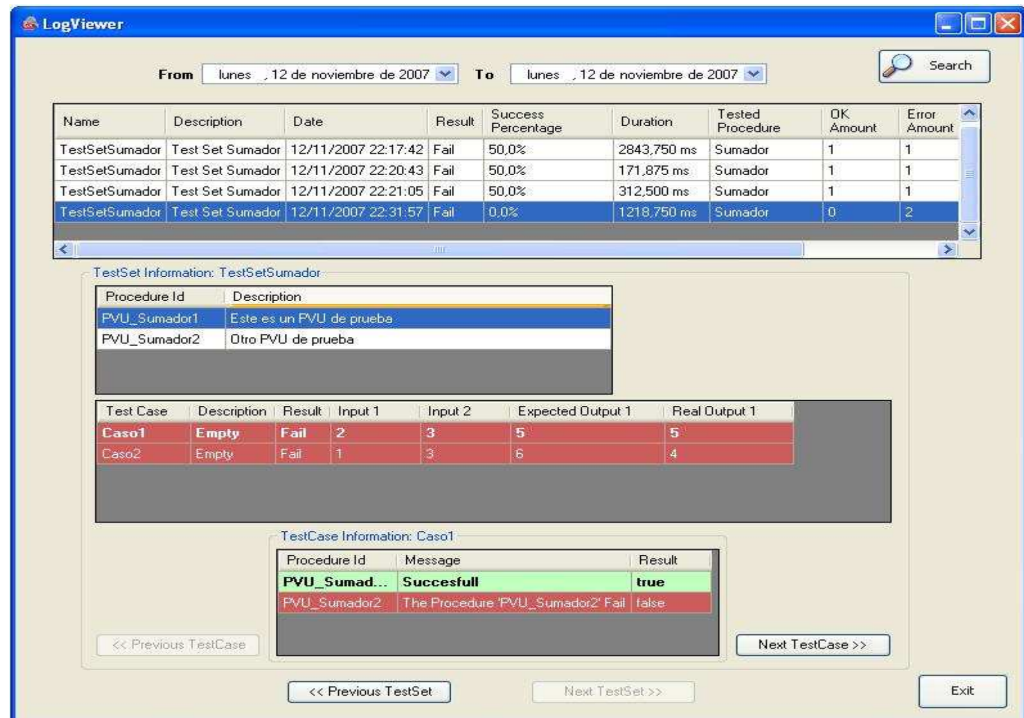


Fig. A- 39: Visor de *Logs* (GXUnit2).

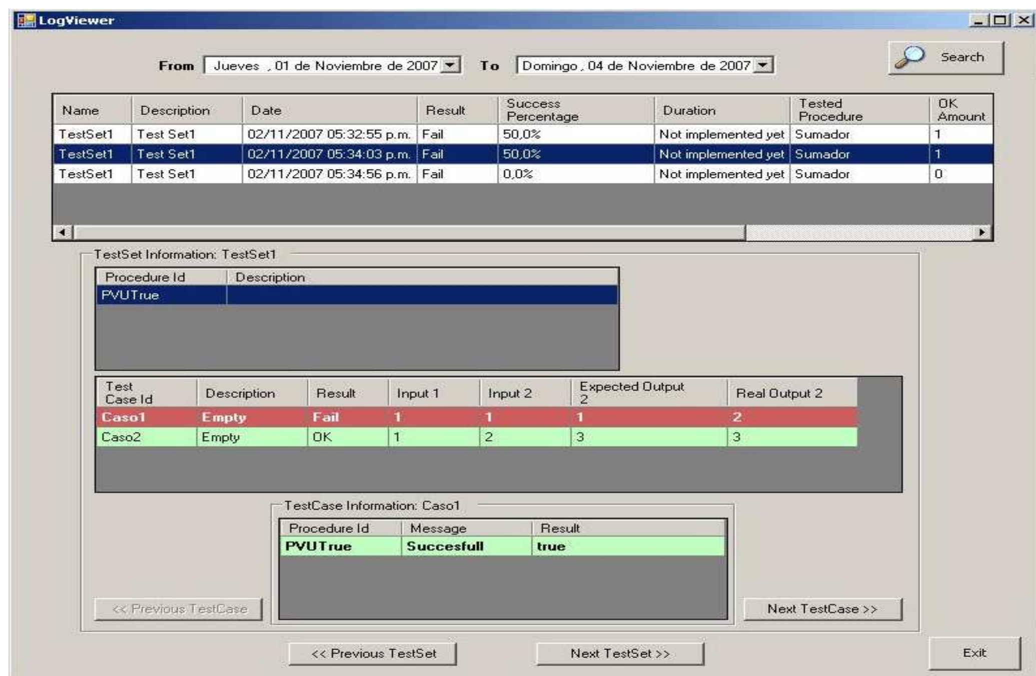


Fig. A- 40: Otra vista del visor de *Logs* (GXUnit2).

La extensión y los objetos TestSet creados son capaces de responder frente a una modificación externa de los procedimientos a verificar o de los PVU. Al modificarse los parámetros de entrada o salida de un procedimiento verificado por un TestSet, los casos de prueba y los PVUs asociados posiblemente dejen de ser válidos. Debe destacarse que la extensión solo detecta cambios en los parámetros y no en el cuerpo de los objetos procedure. Tras modificar los parámetros de un procedimiento a verificar, al abrir el TestSet que lo verifica presenta una ventana similar a la mostrada en la figura A-41. Observando la figura se aprecia que en el cuadrante superior izquierdo de la pantalla se presentan los parámetros que tenía el procedimiento a verificar previo el cambio y debajo los nuevos parámetros. En el cuadrante superior derecho se muestran los antiguos casos de prueba, que perdieron validez tras el cambio de parámetros. Debajo se pueden agregar casos de prueba de forma que contemplen los cambios. ***Los antiguos valores pueden ser reutilizados***, ya sea escribiéndolos manualmente o arrastrándolos con el botón izquierdo del ratón desde la grilla superior a la grilla inferior.

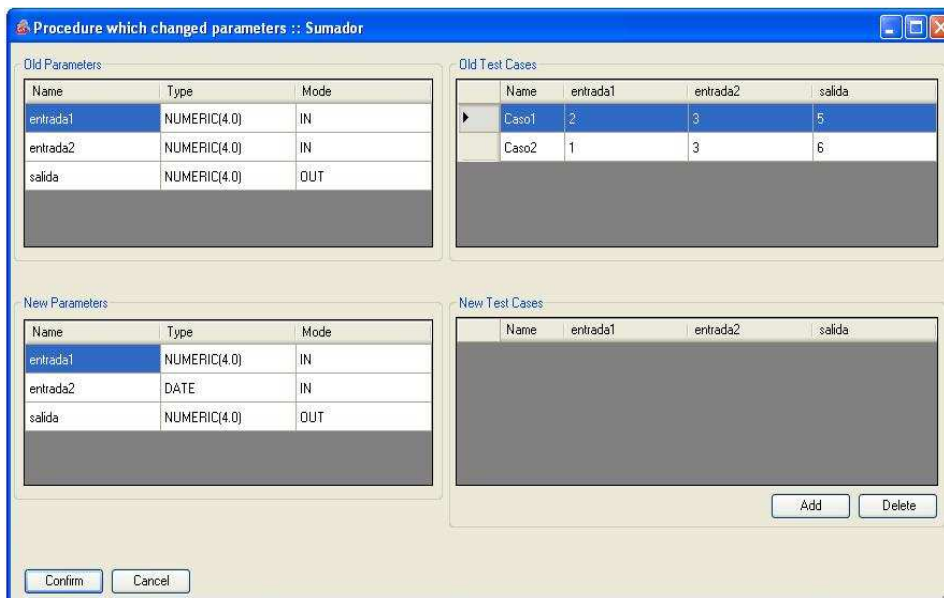


Fig. A- 41: Impacto en el TestSet por cambio en la regla *parm* (GXUnit2).

Los PVUs pueden ser modificados, por lo cual podría ocurrir que un PVU deje de ser válido para verificar un procedimiento dado. Solo se detectan cambios en los parámetros de entrada y salida del PVU. Cualquier cambio en el cuerpo del PVU no tendrá impacto directo sobre el TestSet al que se encuentra asociado. En el caso en que un PVU deje de ser válido para el TestSet, al abrir el objeto TestSet que lo tiene asociado se mostrará una advertencia similar a la brindada en la figura A-42.

Fig. A- 42: Impacto en el TestSet por cambio en PVUs (GXUnit2).

Se destaca finalmente que la extensión permite configurar algunos parámetros, como ser la ubicación de los archivos de log y el nivel de detalle del log. Las figuras siguientes ejemplifican al respecto.

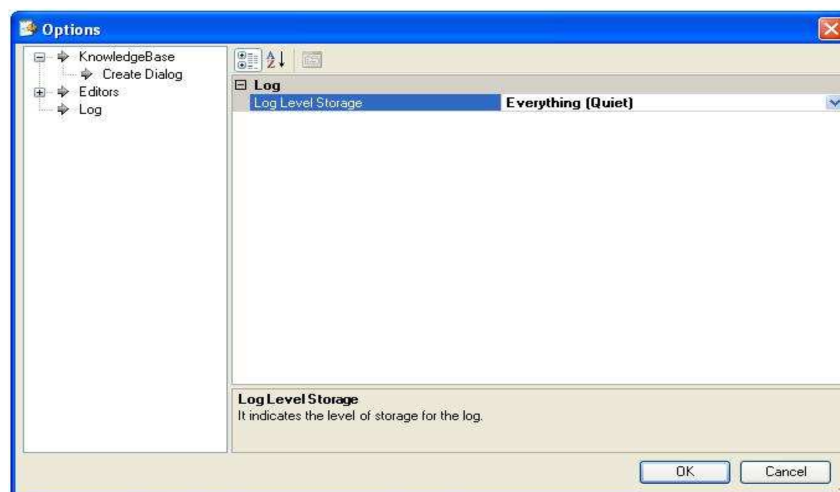


Fig. A- 43: Configuración del nivel de detalle del *log* (GXUnit2).

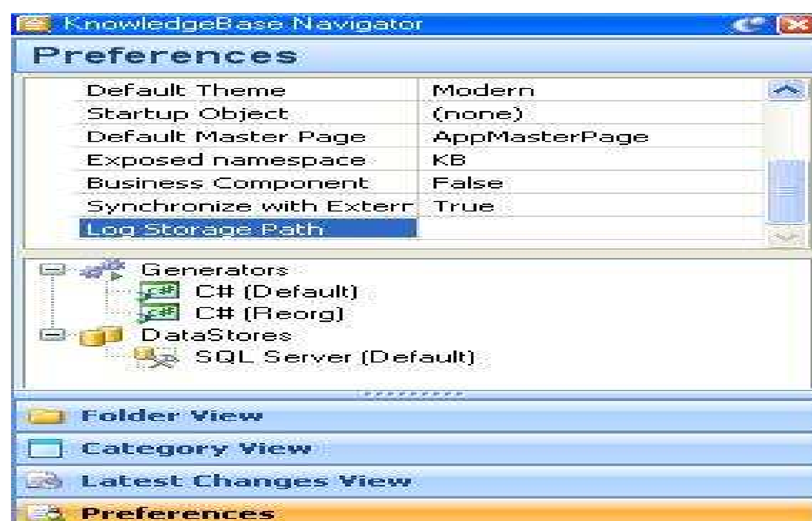


Fig. A- 44: Ruta donde se ubica el *log* (GXUnit2).

Anexo B

E**X**TREME **P**ROGRAMMING (XP)

En este capítulo se expone una descripción abreviada de eXtreme Programming (XP).

Este capítulo está organizado de la siguiente manera:

- En la sección B.1 se ofrece una reseña.
- En la sección B.2 se describe la metodología original XP.
- En la sección B.3 se describe la nueva versión de XP.
- En la sección B.4 se expone un comparativo entre ambas versiones de la metodología.
- En la sección B.5 se ofrece una revisión breve de la bibliografía.

Las secciones B.1 y B.2 se construyen utilizando como base la descripción sobre XP extraída del informe [DA02].

B.1 Reseña

Durante el final de la década de 1980 y principios de los 1990, Kent Beck y Ward Cunningham colaboraron y refinaron sus prácticas con la idea de lograr que el desarrollo de *software* fuera más adaptativo y orientado hacia las personas [Fow05]. Se apoyaron en ideas y prácticas existentes, las cuales van adaptando. Se pueden encontrar en XP notorias influencias de las ideas de Chistopher Alexander, Takeuchi & Nonaka, Jacobsen y otros, las cuales se constituyen en sus raíces [Bec299] [BA04]. En cuanto a las prácticas de XP, Gerald M. Weinberg, según se cita en [LB03], destacó que varias de las técnicas utilizadas en 1957 por un equipo integrado por él, H. Jacobs y otros[86], eran indistinguibles de las propuestas por XP.

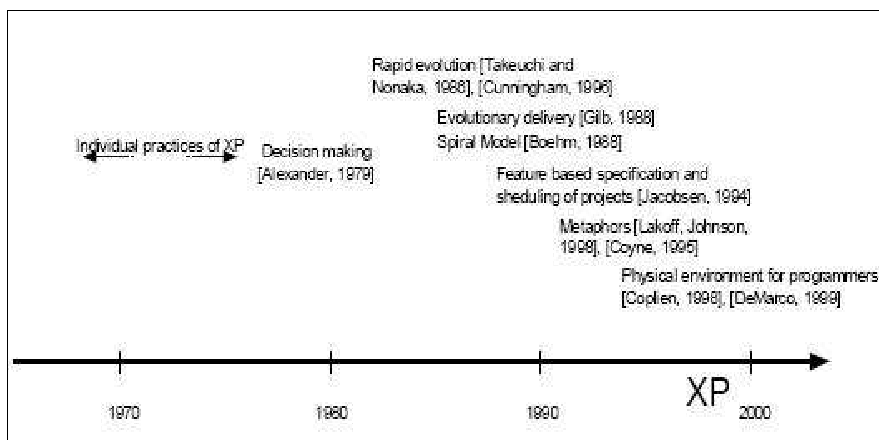


Fig. B- 1: Raíces de eXtreme Programming [Abr02 fig.3].

En Marzo de 1996 Kent Beck fue contratado para revisar el proyecto C3 [BA04] [C304] [Fow107] en Daimler Chrysler. El proyecto fue restaurado bajo su liderazgo y la metodología *eXtreme Programming* (XP) se formalizó [C3T98], colaborando estrechamente en dicha formalización Ward Cunningham y Ron Jeffries.

Kent Beck publicó en 1999 el libro que puede considerarse como el manifiesto de la metodología: "*Extreme Programming Explained: Embrace Change*" [Bec199], habiendo publicado luego, junto a Cynthia Andres "*Extreme Programming Explained: Embrace Change. 2nd. Edition*" [BA04] en el año 2004, donde se ofrece una nueva versión de XP (B.3), luego de varios años de experiencia en su aplicación[87].

Según Kent Beck: "*El desarrollo de software falla en entregar, y falla en entregar valor. Esta falla tiene un enorme impacto tanto humano como económico. Necesitamos encontrar una nueva manera de desarrollar softwar*e" [Bec199].

[86] Se refiere al equipo que luego se reunificó para el proyecto Mercury, citado en la literatura reiteradas veces como un antecedente de *Test Driven Develpment* [LB03] (3.3).
[87] En este capítulo se expondrán ambas versiones y un comparativo entre las mismas.

Beck plantea cuatro variables a controlar en todo proyecto de desarrollo de *software* [Bec199]:

- Costo
- Tiempo
- Calidad
- Alcance

La fuerza externa al proyecto, en este modelo, elegirá tres de las cuatro variables a controlar, y el equipo de desarrollo la remanente.[88] Afirma que la variable alcance es la que más preocupación debe suscitar; administrándola es posible controlar las variables de costo, tiempo y calidad. El alcance es un variable con alta variación, ya que los requerimientos nunca están claros al comienzo y los usuarios no logran explicar cabalmente lo que desean. Dado que el proyecto cambiará de dirección habitualmente, el proceso de desarrollo debe ser altamente tolerable al cambio. En cuanto a la calidad, es la variable menos libre, en la que no serían tolerables desviaciones que la aparten de un nivel de excelencia. Beck y Fowler [BF00] alertan sobre una calidad externa, visible para el cliente, y una calidad interna, visible al Analista, la cual tiene el más fuerte impacto sobre la velocidad de desarrollo y la calidad en general. Disminuyendo la calidad interna puede obtenerse un breve incremento de velocidad, que será rápidamente seguido por un notorio decaimiento en la misma.

Para resolver este tipo de problemas propone la sustitución del modelo "en cascada"[89] por un modelo de desarrollo donde se da prioridad a la liberación periódica de programas que ofrezcan valor para el cliente, construyendo el sistema de forma incremental en base a períodos cortos de tiempo (iteraciones) que agregan funcionalidad, apoyándose fuertemente en las pruebas y la reconstrucción (refabricación) de código [Bec199] [BF00]. Dicho proceso iterativo e incremental propuesto por XP se grafica en la figura B-2.

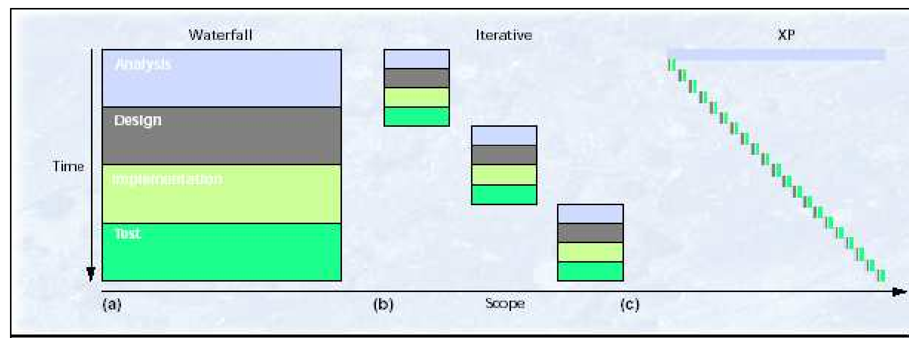


Fig. B- 2 (a) Modelo en cascada, (b) Modelo iterativo (ej.Espiral), (c) XP [Bec299].

[88] Si por el contrario, en un proyecto de desarrollo la fuerza externa intenta fijar todas las variables entonces se irá seguramente en desmedro de la calidad.

[89] "*Waterfall model*" [Wat02]: Basado en el modelo propuesto por Royce en 1970 en la publicación "*Managing the Development of Large Software Systems: Concepts and Techniques*". Puede verse como evolución del modelo de 9 fases de Bennington expuesto en 1956 en la publicación "*Production of Large Computer Programs*".

Sostiene que, bajo ciertas condiciones, es posible mantener 'constante' el costo del cambio[90]: "*Una de las asunciones universales de la ingeniería de software prescribe que el costo de cambiar un programa crece exponencialmente durante el tiempo… El problema es que esta curva ya no es válida… con cierta combinación de tecnología y prácticas de programación, es posible experimentar una curva que es lo opuesto*" [91] [Bec199]. En la figura B-3 se muestra la superposición de ambas curvas.

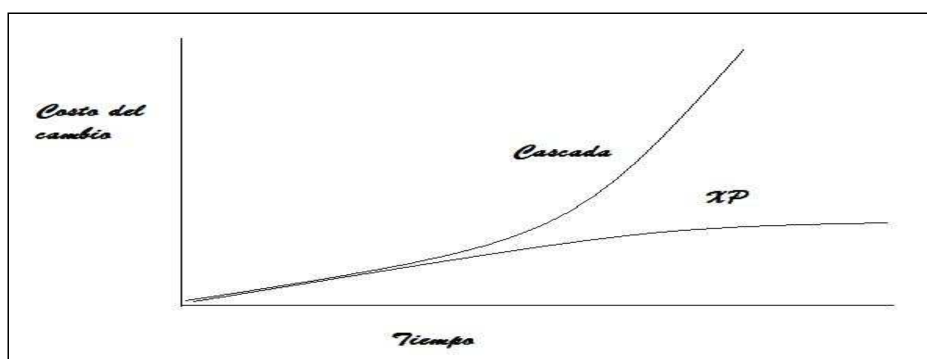


Fig. B- 3: Costo del cambio.

Lo anteriormente expuesto se refiere al costo del cambio. No significa que el costo de enmendar errores (DCI) [92] no crezca exponencialmente con el tiempo, todo lo contrario según la evidencia empírica. Se pone entonces énfasis en actividades de prueba, las cuales caracterizan a la metodología; al punto de enfatizarse que en XP las actividades de prueba son tan importantes como las de programación [BA04].

B.2 Descripción original de XP

B.2.1 Definición

"*eXtreme Programming es una metodología ágil, para equipos de desarrollo pequeños a medianos, enfrentados a desarrollar software en base a requerimientos vagos o rápidamente cambiantes…es una forma liviana, eficiente, de bajo riesgo, predecible, científica y divertida*[93] *de desarrollar software*".

Kent Beck [Bec199].

XP fue pensado teniendo en mente equipos de no más de veinte desarrolladores [Bec03], originalmente entre dos y diez [Bec199]. El problema de la escalabilidad, tanto en el sentido del tamaño del equipo en cuanto a la cantidad de integrantes o en otras dimensiones como la dispersión geográfica y desarrollo de sistemas muy

[90] Curva descubierta por TRW, IBM, Bell Labs Safeguard en los 70´s y expuesta junto a otros resultados en 1981 en el libro *Software Engineering Economics"* según Boehm B [BT03].

[91] Esta fue una de las controversiales opiniones de Kent Beck. Alistair Cockburn, en "*Reexaming the Cost of Change Curve*" [Coc00], propone una visión conciliadora entre esta proposición y lo expuesto por Boehm en "*Software Engineering Economics*". Boehm & Turner en [BT03] comentan acerca que en grandes proyectos se seguirían obteniendo valores similares a la curva original.

[92] DCI: *Defect Cost Increase.*

[93] XP ha sido descrito utilizando también cierta analogía con un juego.

críticos (soporte de vida), plantea aún grandes desafíos. En "*Extreme Programming Explained: Embrace Change. 2nd. Edition*" [BA04], donde se presenta la nueva versión de la metodología que se tratará en (B.3), Beck y Andres abordan los problemas de escalabilidad; proponiéndose soluciones, sin sistematizarlas, para adaptar XP a esas situaciones[94], para las cuales no sería incompatible, incluyendo sistemas críticos para la vida. En el artículo "*Scale-Free Extreme Programming*" [Bec03] Beck trata el tema de la escalabilidad desde un punto de vista teórico, apoyándose en las ideas de Barabasi[95] et al. acerca de redes "*scale-free*"[96]. Otras propuestas, como ser "*Industrial XP*" [IXP07] y "*Distributed XP*"[97] [DXP02], han surgido para intentar una respuesta a algunos de estos retos.

XP original está fundado en cuatro valores, quince principios y doce prácticas.

B.2.2 Valores

Se postula que los cuatro valores esenciales para la mejora de cualquier proyecto son los siguientes [Bec199]:

- **Comunicación:** El equipo a cargo del proyecto debe mantenerse comunicado, intercambiando y colectivizando conocimiento y experiencia, problemas y soluciones a los mismos.
- **Simplicidad:** Hacer lo más simple que pueda funcionar. Diseñar el sistema para obtener la funcionalidad que se necesita en ese momento y cambiarlo a futuro para adaptarlo a nuevas demandas.
- **Retroalimentación:** Se necesita retroalimentación concreta acerca del estado del sistema, funcionando la misma a diferentes escalas de tiempo (minutos, días, semanas y meses).
- **Coraje:** Es el soporte para el resto de los valores, que deben estar implantados para combinarse con el.

B.2.3 Principios

"*De los cuatro valores se derivan doce principios para guiar en el nuevo estilo*" [Bec199]. Estos principios se establecen como un vínculo entre los valores y las prácticas que deberán seguirse. Al presentarlos, Beck divide los principios en fundamentales y otros:

B.2.3.1 Principios fundamentales

- Rápida retroalimentación.

[94] Entre otras propuestas recomienda incluir prácticas como la trazabilidad y subdivisión de los problemas.
[95] "*Linked: How Everything Is Connected to Everything Else and What It Means*" de Barabási, A., Editorial Plume, 2003.
[96] Redes que siguen una distribución con abundancia de nodos con pocos vínculos, y unos pocos nodos con enorme cantidad de vínculos. Ejemplo: Internet.
[97] Propuesta metodológica elaborado por Kircher et al. a partir de la experiencia del proyecto TAO y otros (http://www.cs.wustl.edu/~schmidt/TAO-overview.html)..

- Asumir simplicidad.
- Cambio incremental.
- Abrazar el cambio.
- Trabajo de calidad.

B.2.3.2 Otros

- Enseñar aprendiendo.
- Inversión inicial pequeña.
- Jugar para ganar.
- Experimentos concretos.
- Comunicación honesta y abierta.
- Trabajar a favor de los instintos de los miembros del equipo.
- Responsabilidad aceptada.
- Adaptación.
- Viajar ligero.
- Mediciones honestas.

B.2.4 Ciclos de vida

Las fases de un proyecto XP son las siguientes [Bec199]:

- **Exploración:** Los clientes describen las funcionalidades que quieren implementar bajo la forma de **relatos**[98] **(*stories*)**. Los relatos son muy breves descripciones de funcionalidad visible por el usuario con el mínimo volumen de información necesario para describirla. El resto de los integrantes del equipo de desarrollo se familiarizan con las herramientas, arquitectura y se va construyendo la **metáfora**[99] o visión en común del sistema. Es en esta fase que también se crean pequeños programas **(*spikes*)** con el objetivo de explorar potenciales soluciones y encontrar respuestas a problemas técnicos o de diseño.
- **Planificación de versiones:** A partir de los relatos los desarrolladores deciden el conjunto inicial de los mismos a implementar para la primera versión, escogiendo el mínimo conjunto que tenga sentido. En siguientes versiones será el cliente quien seleccione los relatos a implementar. La primera versión es crítica pues se desarrolla mientras el sistema a producir aún “no respira”. Se realizan reuniones con los clientes para la elaboración del plan. El tiempo de desarrollo de cada relato se mide en semanas y el plan se explicita colocándolo en un lugar visible a todos.
- **Iteraciones:** El objetivo de las iteraciones es poner en marcha una **versión** **(*release*)**. En varias iteraciones se completará una versión Los

[98] Pueden interpretarse como pequeños casos de uso expresados en fichas (*cards*) que describen un requerimiento o una funcionalidad de forma breve. Deben poder ser programados en una o dos semanas. Se les asiga una prueba de aceptación.
[99] Refiere a la visión común de todos los integrantes del equipo, incluyendo a los clientes, acerca del sistema a implementar y su dominio.

desarrolladores subdividen los relatos en **tareas (*tasks*)** y se estiman los tiempos de cada tarea. Cada tarea debe llevar pocos días en ser completada y se asigna a parejas de programadores quienes escriben las pruebas y codifican los módulos. Al completarse cada tarea se integra al código existente y se corren las pruebas de regresión. Mientras tanto los clientes especifican las **pruebas de aceptación**.

- **Producción-Mantenimiento:** En esta fase se hacen iteraciones más breves a los efectos de certificar que el sistema está listo para entrar en producción. Esto último podrá ocurrir recién luego de alcanzada la aprobación del cliente. A partir de la entrada en producción se continuará con las iteraciones en caso de necesitarse nueva funcionalidad. El equipo de desarrollo deberá encargarse, a partir de ese momento, del sistema en producción así como de la incorporación de esa nueva funcionalidad. El mantenimiento es el estado normal de un proyecto XP.
- **Finalización:** Cuando el cliente decide que no necesita producir nuevos relatos se produce entonces la documentación final que se considere estrictamente necesaria y se da por finalizado el sistema[100].

En la figura B-4, se ilustra XP de acuerdo a las diferentes escalas de tiempo.

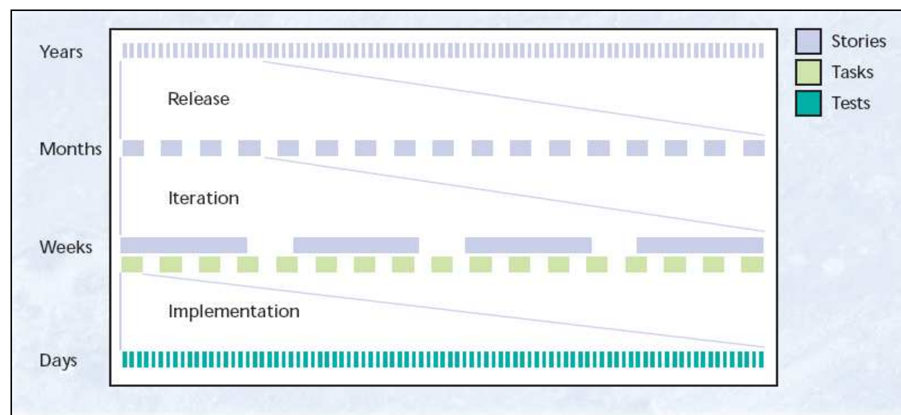


Fig. B- 4: Escalas de tiempo en XP [Bec299].

B.2.5 Prácticas

XP en la versión original [Bec199] comprende veinte prácticas. Si bien muchas de las mismas recogen ideas planteadas por otros autores, tal como se destacó en (B.1), algunas de ellas han generado polémicas.

- **Planificación (*The Planning Game*):** Los desarrolladores estiman el esfuerzo que se necesita para implementar los relatos y el cliente decide que relatos implementar para cada versión.

[100] También se considera el hecho que un proyecto finalice por haberse abortado, en cuyo caso se recomienda efectuar una etapa de reflexión sobre las razones del fracaso y como corregirlas.

- **Pequeños entregables (*Small Releases*):** Cada versión debe ser tan pequeña como sea posible, y debe contener los requerimientos del negocio más valiosos.
- **Metáfora (*Methapor)*:** El sistema es definido como una "metáfora" que conocen tanto el cliente como el equipo de desarrollo. Se explicita el sistema en forma general y coherente.
- **Diseño Simple (*Simple Design*):** Se busca la simplicidad de diseño. Se elimina todo código extra y complejidad innecesaria. Se diseña por la funcionalidad que ha sido definida y no por la potencialmente necesaria.
- **Prueba (*Testing*):** XP es una metodología fuertemente apoyada en la prueba de programas. El desarrollo es guiado por los casos de prueba. Se hace especial hincapié en pruebas unitarias y pruebas de aceptación, aunque también se menciona la necesidad de otros tipos de pruebas [Bec199]. Las pruebas deben ser automatizadas. Según Beck: "*Cualquier función de un programa sin una prueba automatizada simplemente no existe*" [Bec199]. Los desarrolladores codifican primero los casos de prueba unitarios y luego el código de la aplicación que satisface dichos casos. Se establece un paradigma de "pruebe, luego codifique". Cada caso fallido oficia de punto de ingreso de código nuevo. Los clientes escriben las pruebas de aceptación, asociadas a cada relato. Las pruebas son ejecutadas en forma continua y automatizada. Cada vez que una porción de código es escrita se la somete a todo el conjunto de casos de prueba correspondientes. Esta es una de las prácticas que más caracterizan la metodología, que proclama con ella un muy extensivo uso de la prueba de programas, dando lugar a la formalización de la disciplina de desarrollo denominada TDD (*Test Driven Development*) (3.3).
- **Refabricación (*Refactoring*):** Se reestructura el sistema durante todo el ciclo del proyecto, removiendo redundancias, eliminando funcionalidad obsoleta, mejorando la comunicación, rejuveneciendo diseños antiguos, simplificando para incrementar la calidad. No se traduce en cambios observables en el comportamiento del sistema y la adición de toda nueva funcionalidad debería ser precedida por la refabricación.
- **Programación en parejas (*Pair Programming*):** Dos programadores trabajando con una misma máquina. El desarrollo de las tareas se realiza en parejas de programadores.
- **Propiedad Colectiva (*Collective Ownership*):** Cualquiera dentro del equipo de desarrollo puede cambiar cualquier parte del código en cualquier momento.
- **Integración Continua (*Continuous Integration*):** La integración de código es continua y ocurre varias veces al día. Se integra cada pieza en cuanto está pronta. Todas las pruebas deben pasarse para aprobar la integración.
- **40 Horas Semanales (*40-horas week*):** No se debe trabajar más de 40 horas semanales. Si se necesita tiempo extra más de una semana consecutiva ello debe considerarse un problema a resolver.
- **Cliente en el Lugar (*On-Site Customer*):** Un cliente real debe estar siempre disponible para el equipo de desarrollo, integrado al mismo.

- **Estándares de Codificación: (*Coding Standard*):** El código debe adherir a estándares conocidos por el equipo para así facilitar otras prácticas.

En [Bec299] se mencionan también:
- **Espacio Abierto de trabajo (*Open Space*).**
- **Solo son reglas (*Just Rules*):** El equipo sigue las reglas o puede cambiarlas, pero respetándolas una vez acordadas.

B.2.6 Roles

En XP se definen los siguientes roles [Bec199]:

Conducción
- Administrador (*Big Boss*):
- Encargado de seguimiento *(Tracker).*
- Entrenador *(Coach).*

Resto
- Cliente.
- Consultante.
- Programador.
- Verificador (*tester*).

Según R.Jeffries et al. [Jef00]:

"eXtreme Programming es una disciplina de desarrollo de software con valores de simplicidad, comunicación, retroalimentación y coraje. Nos enfocamos en los roles de cliente, administrador y programador acordando derechos y responsabilidades claves para cada uno de dichos roles."

B.2.7 Círculo de vida y adaptación

XP se presenta como una lista de prácticas, apoyadas en principios y valores. Debe considerarse como una línea de partida. Se comienza como dice "en el libro", se va adaptando a la medida de los proyectos y la experiencia. La guía de un proyecto hacia el éxito se ejemplifica en el circulo de vida que se expone en la figura B-5, donde se muestran los roles y las acciones a cometer[101]. En [Jef00] se concluye que para llevar un proyecto a su más satisfactoria conclusión se debe seguir el círculo: definir, estimar, escoger, construir.

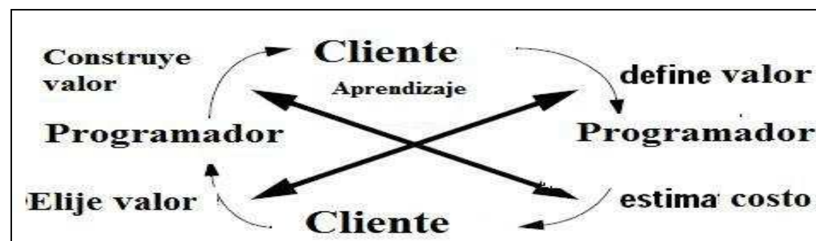


Fig. B- 5: Circulo de vida. Basado en fig. 2.3 de [Jef00].

101 Las acciones están indicadas por flechas curvas que forman el "circulo de vida". El origen de las flechas curvas parte del rol que hace la acción representada por la misma y el destino es el rol al que se destina el resultado. Las flechas con doble sentido representan el proceso de aprendizaje.

B.3 El nuevo XP (XP2)

En la segunda edición del libro *"Extreme Programming: Embrace Change"*[102] [BA04], se plantean cambios y una rearticulación de la metodología. La definición original es complementada indicando que XP se basa en solucionar restricciones en el desarrollo de *software,* siendo aplicable también en equipos de cualquier tamaño y en proyectos con requerimientos firmes, no volátiles. El nuevo XP se basa en cinco valores, catorce principios y veinticuatro prácticas, agrupadas estas últimas en primarias y de corolario; en contraste con los cuatro valores, quince principios y doce prácticas originales. Se aprecia en su formulación una mayor consideración en aspectos organizacionales[103] y de gestión de proyectos. Sin embargo aún mucha de la información que es posible encontrar sobre XP refiere a su versión original [Fow05], especialmente en cuanto a reportes de experiencias. En [Marc05], [Wil05] y [Tos07] se ofrecen resúmenes y cuadros comparativos entre ambas versiones.

B.3.1 Valores fundamentales

- **Comunicación** ver (B.2.2).
- **Simplicidad** ver (B.2.2).
- **Retroalimentación** ver (B.2.2).
- **Coraje** ver (B.2.2).
- **Respeto:** Debajo de la superficie de los anteriores valores subyace el respeto entre los integrantes del equipo y para con el proyecto.

B.3.2 Principios

- **Humanidad:** Los factores humanos son clave para entregar calidad. Se debe lograr balance entre estos factores y los organizacionales. Se deben tener en cuenta las necesidades de los seres humanos.
- **Economía:** Se debe producir valor para el cliente.
- **Beneficio Mutuo.** Todas las actividades deben traer beneficios a todos.
- **Auto-Semejanza:** Reutilizar soluciones similares en contextos diferentes, por ejemplo el patrón "*escribir pruebas y luego el código que pasa la prueba*" aplicado a nivel unitario y de aceptación.
- **Mejora:** Es esencial la mejora continua y hacer el mejor trabajo posible.
- **Diversidad:** De talentos, experiencia y caracteres.
- **Reflexión:** Reflexionar periódicamente sobre el proyecto.
- **Flujo:** Entregar un flujo continuo de valor.
- **Oportunidad:** Transformar los problemas en oportunidades.
- **Redundancia:** Prever la defensa en profundidad. Este principio apoya a prácticas tales como TDD y programación en parejas.

[102] Kent Beck y Cynthia Andres.
[103] En particular la "*teoría de las restricciones*" de Goldrat. Apoyándose en ella se discute al respecto de la disolución de los equipos XP altamente productivos por parte de organizaciones que no pueden funcionar a su ritmo [BA04].

- **Falla:** Mejor es tratar y fallar, que no hacerlo. Todo error se transforma en aprendizaje. Implementar soluciones diferentes en paralelo permite aprender.
- **Calidad:** Es primordial, debe elevarse al máximo, no es un variable de control.
- **Pequeños pasos:** Proceder dado avances pequeños de manera iterativa, previniendo así grandes fallas y permitiendo correcciones de la dirección menos costosas.
- **Responsabilidad aceptada:** Aceptar una orden recibida implica conllevar la responsabilidad sobre la misma.

B.3.3 Prácticas

Las prácticas son divididas en dos grandes grupos, lo que constituye la primera novedad en esta área. Un conjunto de ellas, llamadas prácticas primarias, son las que se recomienda aplicar en primera instancia. Las restantes se ponen en práctica una vez establecidas y ajustadas las primarias. La aplicación de cada práctica implica mejoras en el proceso de desarrollo, la aplicación de varias prácticas implica dramáticas mejoras debido a su interacción [BA04]. La segunda novedad importante es que se presentan veinticuatro prácticas, habiendo desaparecido algunas de las prácticas originales. Las veinticuatro prácticas propuestas se presentarán utilizando una categorización basada en la sugerida por Marchesi [Marc05].

B.3.3.1 Prácticas primarias[104]

Son las prácticas que deben aplicarse primero.

- <u>Planificación y Análisis de Requerimientos</u>
  - **Relatos (*Stories*)** (ver B.2.4)
  - **Ciclo semanal (*Weekly Cicle*)** Iteraciones semanales, con una reunión al comienzo para selección de relatos, división en tareas y revisión.
  - **Ciclo trimestral (*Quartely Cicle*)** Planificación a mayor escala, elección de "temas" los que luego serán completados con relatos.
  - **Holguras (*Slacks*)** Rellenar con tareas que puedan descartarse para contar con holguras.

- <u>Factores humanos</u>
  - **Ubicarse contiguo (*Sit Together*):** Espacio de trabajo abierto.
  - **Equipo completo (*Whole Team*):** Incluir en el equipo personas con todas la habilidades y experiencias requeridas.
  - **Espacio de trabajo informativo (*Informative Workspace*):** Pizarrones y todo lo necesario para informar en forma bien visible.
  - **Trabajo con energía (*Energized Work*):** Limitar la jornada.
  - **Programación en parejas (*Pair Programming*).**

[104] Se ofrece junto al nombre de cada práctica una breve explicación solo de las prácticas nuevas o con variantes.

- Elaboración: Diseño, Codificación, Prueba
  - **Diseño incremental *(Incremental Design*)**
  - **Prueba antes que programa (*Test-First Programming*)**
  - **Construcción en diez minutos (*Ten-Minute Build*):** Automáticamente construir el sistema y ejecutar las pruebas en diez minutos.
  - **Integración continua (*Continous Integration*)**

B.3.3.2 Prácticas de corolario

Estas prácticas se aplican una vez establecidas las primarias.

- Planificación y Análisis de Requerimientos
  - **Involucrar al Cliente Real *(Real Client Involvement).***
  - **Implementación incremental *(Incremental Deployment)*:** Al sustituir un sistema hacerlo mediante implementaciones parciales, sucesivas e incrementales.
  - **Negociar alcance de contratos *(Negotiated Scope Contract)*:** Negociar el alcance, dada la calidad, precio y tiempo. Puede ser útil hacer varios contratos sucesivos de corto alcance.
  - **Pagar por el uso *(Pay per Use)*** Como alternativa de licenciamiento.

- Factores humanos
  - **Continuidad del equipo *(Team Continuity)*:** No dispersar el equipo al final del proyecto.
  - **Achicar el equipo *(Shrinking teams)*:** Al aumentar la productividad disminuir el tamaño del equipo, enviando miembros a otros equipos.

- Elaboración: Diseño, Codificación, Prueba
  - **Análisis raíz-causa (*Root-Cause Analysis*)** Eliminar defectos y sus causas.
  - **Código compartido (*Shared Coded).***
  - **Código y pruebas (*Code and Test*):** Únicos artefactos a preservar.
  - **Versión Única (*Single Code Base*)**
  - **Implementación diaria (*Daily Deployment*).**

B.3.4 Roles

Los roles en un equipo XP maduro no son rígidos y resulta natural compartirlos[105], pero se advierte que si algunos roles no se personalizan, como ser el de arquitecto o administrador, se podría atentar contra la escalabilidad[106] [BA04] [Bec03].

---

[105] XP no recomienda la profunda división de labores ni la separación muy marcada de los roles, incentivando de esta forma la descentralización de decisiones.
[106] De ser cierta la presunción que equipos XP podrían crecer como redes “*scale free*”.

Los roles considerados son los siguientes [BA04]:

- Administrador del Proyecto.
- Administrador del Producto.
- Diseñador de interacción.
- Ejecutivo.
- Entrenador
- Escritor técnico.
- Evaluador.
- Programador.
- Usuario.

B.4 Correlación de prácticas XP vs. XP2

En la tabla B-1 se ofrece una posible correlación entre prácticas, basada en la propuesta por Marchesi [Marc05].

XP2 o Nuevo Xp	Xp original
Ubicación contigua	[Expuesta en Bec299]
Equipo completo	
Espacio informativo	Juego de Planificación
Trabajo con energía	40 horas semanales
Programación en parejas	Programación en parejas
Relatos	Juego de planificación
Ciclo semanal	Juego de Planificación/ Pequeños entregables
Ciclo trimestral	Juego de planificación/ Pequeños entregables
Holguras	Juego de planificación
Construcción c/10'	
Integración continua	Integración continua
Cód. prueba antes que programa	Prueba
Diseño incremental	Diseño Simple/Reconstrucción
	Metáfora
Código compartido	Propiedad colectiva
Involucrar al cliente	Cliente en el lugar
	Estándares
Continuidad del equipo	
Achicar el equipo	
Análisis de causa	
Código y prueba	Diseño simple
Única versión base	
Implementación diaria	
Negociar alcance de contratos	
Pagar por uso	
Implementación incremental	Pequeños entregables

Tabla B- 2: Correlación de prácticas entre XP y XP2.

Además de las nuevas prácticas, muchas de las cuales surgen como desdoblamiento de prácticas anteriores, es notoria la desaparición de la práctica “metáfora”[107], de los estándares de codificación[108], el cambio en el límite de 40 horas semanales que pasa

[107] Si bien se mantienen menciones a la misma, por ejemplo al describir el rol de “*Interaction Designer*”.
[108] Considerado como algo obvio en la nueva versión.

a quedar a criterio de los involucrados, pero manteniendo la filosofía de no excederse, y la practica de reconstrucción, que aparece dentro de 'diseño incremental'.

En cuanto a los roles se destaca que el rol de entrenador ya no se prescribe aunque se recomienda, apareciendo claramente diferenciados el rol del administrador del proyecto y del producto, así como el rol del arquitecto y de los escritores técnicos que se encargan de manuales y documentación.

B.5. Consideraciones

La metodología eXtremme Programming ha tenido amplia difusión desde su aparición, siendo probablemente una de las metodologías ágiles que más interés ha despertado tanto en ámbitos de la industria como académicos, y el más ampliamente reconocido método ágil [BT03]. Las aseveraciones realizadas por sus proponentes, especialmente Kent Beck, referidas al costo del cambio y las prácticas, en particular TFP, reconstrucción y programación en parejas, han suscitado discusiones y polémicas desde su introducción. En la literatura es posible encontrar mucha información, aunque la evidencia empírica acerca del éxito de su aplicación no puede considerarse concluyente[109]. Dada esta situación se resumirán consideraciones acerca de sus prácticas; siendo de destacar que además son de aplicación también las consideraciones efectuadas en el análisis sobre TDD[110] (3.3.2).

Beneficios:

- **El desarrollo iterativo e incremental** es considerado como buena práctica dentro de la ingeniería de software. XP lo lleva a sus extremos con iteraciones sumamente breves orientadas al desarrollo de funcionalidades puntuales con valor para el cliente; controlando el alcance y maximizando la calidad.
- **El énfasis en pruebas unitarias automatizadas y de aceptación** no solo es altamente beneficioso en prevención de defectos sino que muy probablemente se constituyó en el más efectivo vehículo para la divulgación de prácticas de prueba unitaria y su aplicación.
- **La programación de a pares**, como facilitadora de la aplicación de TDD y de prácticas beneficiosas (revisiones entre pares) por su impacto en la disminución de la tasa de defectos.
- **Los factores humanos**, con el énfasis puesto en la comunicación, la continuidad del equipo, el trabajo no agobiante.
- **La visualización del proyecto**, mediante la utilización del espacio informativo, compartido y las reuniones periódicas.

[109] Incluso algunos de los proyectos emblemáticos, como ser C3, han sido puestos en duda en cuanto a las aseveraciones acerca de su éxito [SR03], [Kee02].
[110] Dado que las prácticas de XP "*Test First Programming*" y "*Refactoring*" dan lugar a la disciplina.

- **Fuerte involucramiento del cliente** y su participación activa tanto en la definición de los requerimientos mediante relatos, su priorización y la definición de las pruebas de aceptación.
- **Los relatos,** que permiten al cliente especificar de una forma natural sus requerimientos, expresándolos en su dominio del discurso, sirviendo de base para la discusión conjunta con los desarrolladores.
- **Prácticas definidas que introducen disciplina.**

Debilidades

- **Escalabilidad:**[111]
  - **Dificultad para la aplicación en equipos grandes**[112], dadas las prácticas de XP y los factores humanos; si bien se han efectuado aportes en tal sentido, como ser en [Bec03], [BA04], [IXP07].
  - **Dificultad para la aplicación en equipos distribuidos**, dadas las características de las prácticas de XP, si bien se han efectuado experiencias de utilización en equipos distribuidos [KL00] y aportes metodológicos al respecto, como ser en [BA04] y [DXP02].
- **Carencias en la gestión de los proyectos,** si bien se han introducido aportes al respecto, como ser en [Jef00] y [BA04].

Es de destacar que podrían fortalecerse estas y otras debilidades mediante la incorporación de las prácticas de XP bajo metodologías más predictivas [BT03] [SR03] o bajo otras metodologías ágiles con mayor hincapié en la gestión y con la elaboración de mejores y más versátiles herramientas para pruebas.

La metodología XP ofrece ventajas importantes al introducir principios basados en la comunicación, prácticas claras y simples, un alto nivel de disciplina, métodos de programación alineados con el método científico [Mug03] (3.3), promoviendo código limpio y con menos defectos así como pruebas automatizadas. Siendo además especialmente indicada para equipos pequeños y dada la conformación de la mayoría de los equipos de desarrollo en Uruguay, se entiende que debe propiciarse su investigación y difusión, la realización de experimentos controlados así como la recolección sistematizada de datos acerca de experiencias. La elaboración de herramientas para prueba unitaria y de aceptación automatizadas, adaptadas a las herramientas de desarrollo de alto nivel y las interfases de usuario, se vuelve particularmente importante para la aplicación de las prácticas. Por último se entiende que se debe considerar previamente a cada proyecto si aplica al mismo; de aplicar, adaptarla, recogiendo las lecciones aprendidas y siendo cuidadosos en los extremos, como ser en lo referido a la automatización de todas las pruebas.

[111] Podrían considerarse estos aspectos fuera del alcance de XP, ya que originalmente fue concebido para equipos pequeños y no distribuidos, ni para la elaboración de sistemas de soporte de vida.
[112] Al respecto Boehm señala que el límite estaría en equipos con 20 personas [BT03].

B.6. Revisión bibliográfica

A modo de resumen de la revisión bibliográfica efectuada se destaca que en [Bec199] y [Bec299] se efectúa la formalización original de la metodología XP y en [BA04] se la revisa luego de varios años de experiencia por parte de su proponente principal (Kent Beck), planteándose una rearticulación con más énfasis en el control de proyectos y los aspectos organizacionales. En [Abr02] se la describe, analiza y compara con otras metodologías ágiles; mientras en [Lbo04] se la resume y ofrece un estudio comparativo con la norma ISO 9001:2000, describiéndose experiencias. En [Kee02] y [SR03] se exponen visiones críticas [113], estudiándose en [SR03] la aplicación de sus prácticas dentro de otras metodologías. En [BT03] se analiza la curva del cambio, se compara con TSP y se postula lograr un balance entre agilidad y disciplina mediante la complementación con otras metodologías[114]. En [BF00] se profundiza en la planificación y seguimiento (*tracking*) de los proyectos[115] y las prácticas asociadas en tanto que [Jef04] brinda una crónica de un proceso en estilo XP incluyendo un análisis retrospectivo del mismo. En [Tos07] se proponen métricas para el seguimiento en metodologías ágiles, validando con casos de estudio en XP y sugiriendo un catálogo de métricas para XP. En [CH02] se describe el rol del verificador (*tester*) y su vinculación en las etapas del ciclo de vida[116]; mientras en [Jef00] se abordan las prácticas en cada etapa del ciclo de vida. Por último, se destaca a los sitios[117] [Jef07], [Wel06] y XPoogle [XPG07].

[113] Los autores reconocen, sin embargo, valor a varias de las prácticas de XP y a los métodos ágiles.
[114] Boehm describe un día de vida de un proyecto con TSP y con XP, indicando ciertas similitudes fuertes entre ambos. Postula en base a experiencias sobre la complementariedad con metodologías más predictivas y enuncia una guía para lograr el balance.
[115] Contiene además un breve capítulo sobre contratos.
[116] Incluye también un capítulo sobre localizaciones remotas y la inserción en grandes proyectos.
[117] Sitios de Ron Jeffries, uno de los creadores de XP, y Don Wells, participante en el proyecto C3 (B.1).

A n e x o   C

FitNesse

En este capítulo se expone una descripción de FitNesse, herramienta para pruebas automatizadas de aceptación basada en FIT (5).

Este capítulo está organizado de la siguiente manera:

- En la sección C.1 se ofrece una introducción a FitNesse.
- En la sección C.2 se resume acerca de su arquitectura.
- En la sección C.3 se describe acerca de la integración con otras herramientas.
- En la sección C.4 se resumen consideraciones sobre su uso.

C.1 Introducción

FitNesse es una herramienta de colaboración para el desarrollo de *software* y un marco ágil, abierto, que facilita la definición colaborativa de pruebas de aceptación y su ejecución, permitiendo publicar los resultados de forma visible al grupo de trabajo. También es un *wiki*, lo cual permite fácilmente editar y crear páginas conteniendo la definición de las pruebas, y un servidor *web* que no requiere configuración ni inicialización [MMW07].

Según la descripción expresada en el libro "*Fit for Developing Software: Framework for Integrated Tests*" [MC05]:

"FitNesse permite ver, cambiar y crear páginas web conteniendo documentación y tablas para pruebas que pueden ser ejecutadas por FIT. Facilita el acceso a dichas páginas al hacerlas disponibles a navegadores web…".

Según explica Adzic [Adz107] la combinación de FIT con FitNesse brinda la posibilidad de escribir y entender fácilmente pruebas complejas, necesitándose solo una capa delgada de código para implementarlas.

Presenta la particularidad de estar basada en dos trabajos de Ward Cunningham: FIT y *wiki*. Fue desarrollada por Mica Martín, Robert Martín y otros, en lenguaje Java y utilizando TDD como técnica de desarrollo [Mar07][118] habiendo sido liberada para su uso público con licencia GNU-GPL una primera versión en el año 2003.

La herramienta está pronta para utilizar un vez que se desempaqueta el archivo comprimido accesible en Internet[119] [MMW07]. Tal como se expresó no es necesaria configuración ni inicialización para comenzar a utilizarla[120] y posee una completa ayuda en línea. Una vez que se la ejecuta desde una ventana de comandos, levanta un servidor de páginas *web*, accesible tanto localmente como remotamente desde un navegador. En dicho servidor reside un *wiki* que contendrá las páginas con la documentación y las tablas con casos de prueba. Las tablas serán recorridas, página por página, de arriba hacia abajo y de izquierda a derecha. El usuario podrá visualizar los resultados, así como modificar el contenido de las páginas. Todas las páginas tienen ciertas propiedades que pueden ser alteradas. Una de las propiedades permite marcar páginas indicando que las mismas contienen casos de pruebas, lo cual permitirá lanzar su ejecución desde ellas, habilitando un botón en las propias páginas para tal fin. Otra propiedad permite indicar si la página corresponde a un conjunto de pruebas (*Suite*).

[118] R. Martín en [Mar07] describe su experiencia utilizando TDD para la construcción de FitNesse, explicando que contiene cerca de 45.000 líneas de código, de las cuales aproximadamente la mitad corresponden a pruebas unitarias. Relata que fueron capaces de ejecutar todo el conjunto de pruebas muy rápidamente, lo que les permitió proceder en forma incremental, efectuando cambios sin mayor temor y con un muy rápido ritmo TDD, produciendo un producto final confiable. Si bien reconoce que TDD no convirtió a FitNesse en una utopía de diseño, enfatiza que se logró generar un código altamente desacoplado con una configuración simple e interfase intuitiva.

[119] http://www.FitNesse.org/FitNesse.Download.

[120] Si es necesario tener instalado Java 5 o 6.

A continuación se desarrollará un breve ejemplo de utilización; la documentación detallada y guías de utilización pueden encontrarse, entre otros, en [Adz107], [MC05], [MMW07] y en la documentación que se descarga desde Internet junto a la herramienta.

- Se levanta el servicio desde una ventana de comandos ejecutando *java -cp fitnesse.jar fitnesse.FitNesse -p 8081* (el modificador –p permite cambiar el puerto de escucha, en este caso 8081). La figura C-1 muestra la ventana de comandos y la página de inicio cargada al apuntar el navegador hacia la dirección \\http:localhost:8081.

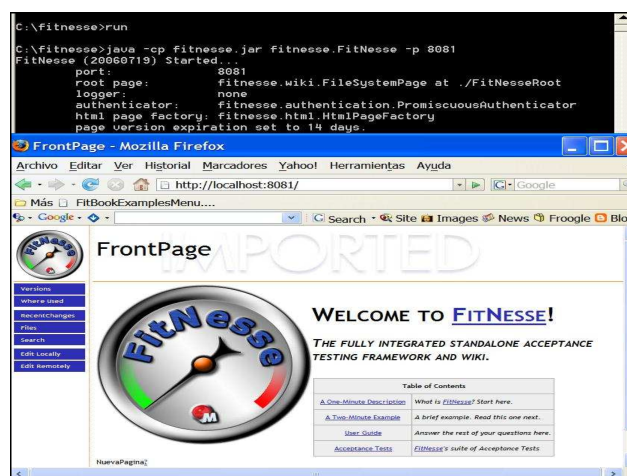


Fig. C- 1: Página inicial de FitNesse.

- Se apunta el navegador a una página inexistente dentro del servidor. Las figuras C-2 y C-3 muestran los pasos para crear la página "NuevaPg" obtenida apuntando el navegado hacia HTTP: \\localhost:8081\NuevaPg y pulsando el enlace rotulado "*create this page*".

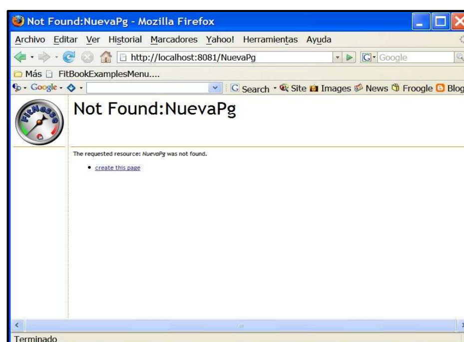


Fig. C- 2: Creación de una nueva página.

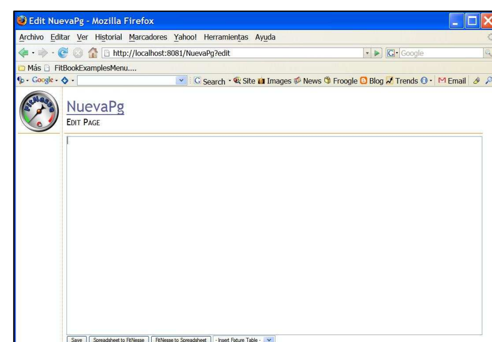


Fig. C- 3: Página nueva, lista para el ingreso de información.

- Las figuras C-4 a C-9 muestran los sucesivos pasos a dar para introducir en la página recientemente creada un vínculo a otras páginas conteniendo los

ejemplos[121] brindados en el libro "*Fit for Developing Software*" [MC05] y adaptados por Vlagsma [Vla06].

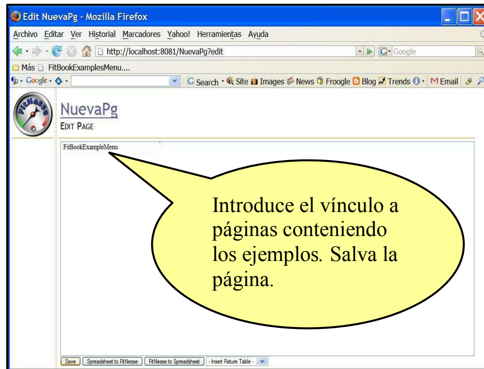


Fig. C- 4: Editando una página para indicar vínculo a páginas de ejemplo.

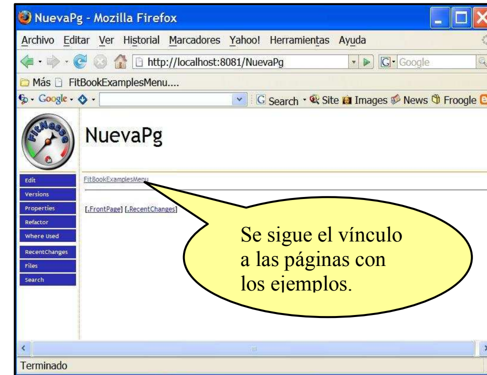


Fig. C- 5: Página pronta para ser usada.

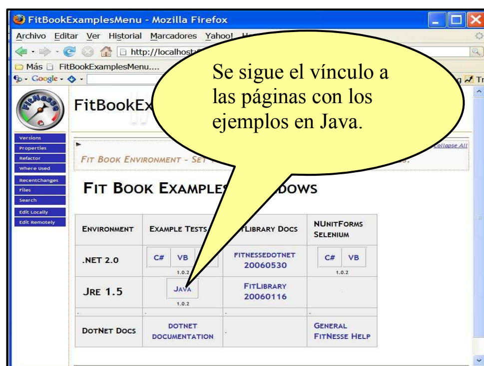
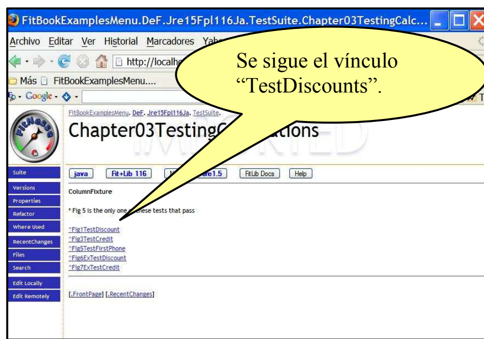


Fig. C- 6: Página inicial de ejemplos.

Fig. C- 7: Página con conjunto (suite) de casos de prueba.



F
Fig. C- 8: Nueva página con conjunto de conjuntos de casos de prueba.

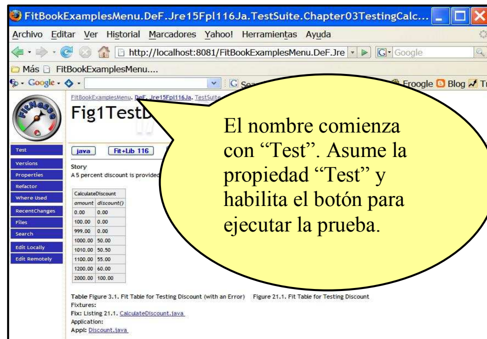


Fig. C- 9: Caso de prueba "Descuentos".

- Se edita la página para modificar el contenido de una fila de la primera fila de la tabla. Tal como se observa en la figura C-10 la página no solo contiene la tabla sino también comandos, comentarios y código JavaScript[122] [JavS07];

[121] Dichas páginas, en el caso del ejemplo, se encuentran en un subdirectorio de la instalación.
[122] Actualmente es una implementación del estándar EcmaScript ECMA-262 aprobado también como estándar ISO/IEC 16262. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

en este caso para mostrar desde la página los fuentes de las *fixtures* que ejecutan las pruebas y el código del SUT a probar.

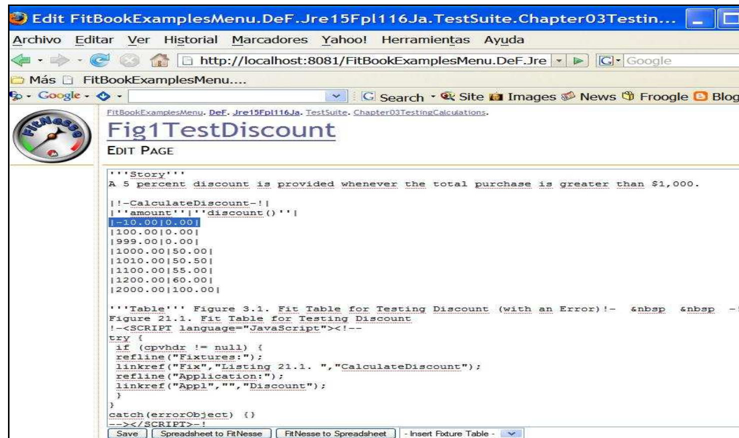


Fig. C- 10: Edición de un caso de prueba.

- La página, una vez guardada, luce como muestra la figura C-11. Las *fixtures* para ejecutar la prueba existen, han sido compiladas y están accesibles[123] por FitNesse, al igual que el SUT. Se ejecuta la prueba pulsando el botón "*Test*". El resultado se muestra en la figura C-12, donde la excepción resaltada en amarillo corresponde al valor negativo de entrada ingresado en la primer fila, según se marcó en la figura C-10. Los casos satisfactorios se resaltan en verde.

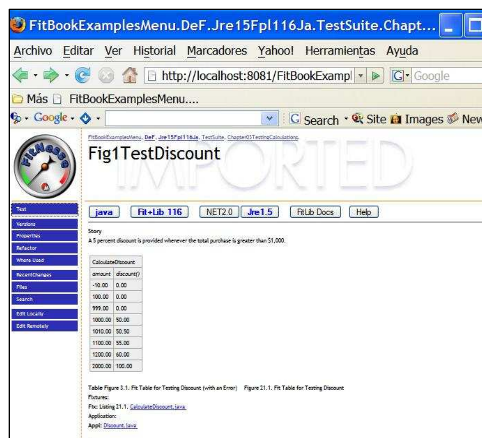


Fig. C- 11: Caso de prueba "Descuentos", modificado.

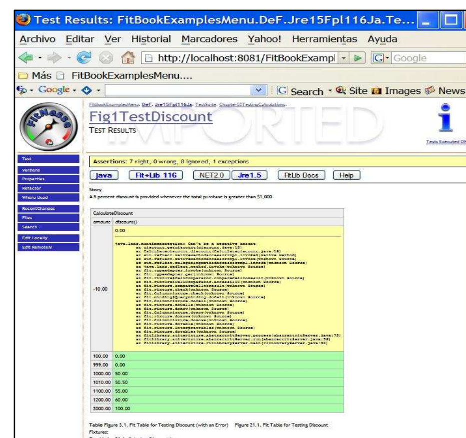


Fig. C- 12: Resultado de ejecutar el caso de prueba "Descuentos".

FitNesse, además de las funcionalidades del *wiki*, posee un lenguaje de marcadores para dar formato a las páginas e indicar acciones, permitiendo utilizar variables para parametrizar las páginas. En las celdas de las tablas se pueden escribir operadores (se asume igualdad), para indicar la comparación a realizar entre los resultados obtenidos y esperados. Es posible cargar en variables, llamadas símbolos, los

[123] Se proveen comandos para definir la ubicación de las *fixtures* y el SUT así como el programa de ejecución (*runner*) a utilizar.

resultados obtenidos para usarlos como entradas posteriormente. Es dable contar con interpretes (*cell handlers*) para los diferentes tipos de datos contenidos en las celdas.

Su *wiki* tiene la particularidad de ser jerárquico, por lo que cual es posible definir "*sub-wikis*"[124], los cuales representan una completa jerarquía de páginas que habitan debajo de otras. Pueden transformarse en conjuntos de pruebas (*suites*) mediante la habilitación de la propiedad correspondiente en su página principal.

Existen páginas especiales para expresar acciones de inicialización (*setup*) y de restauración (*teardown*), que serán ejecutadas respectivamente antes y después de ejecutar las pruebas de cada página, o antes y después, respectivamente, de ejecutar todo el conjunto de pruebas.

C.2 Arquitectura

Consta de un repositorio para almacenar las páginas con los casos de prueba; un servidor *web* y editor *wiki* para presentarlas, editarlas y mostrar los resultados de su ejecución; un programa para vínculo con el servidor FIT y un conjunto de *fixtures* estándares. En la siguiente figura se ilustra acerca de dicha arquitectura.

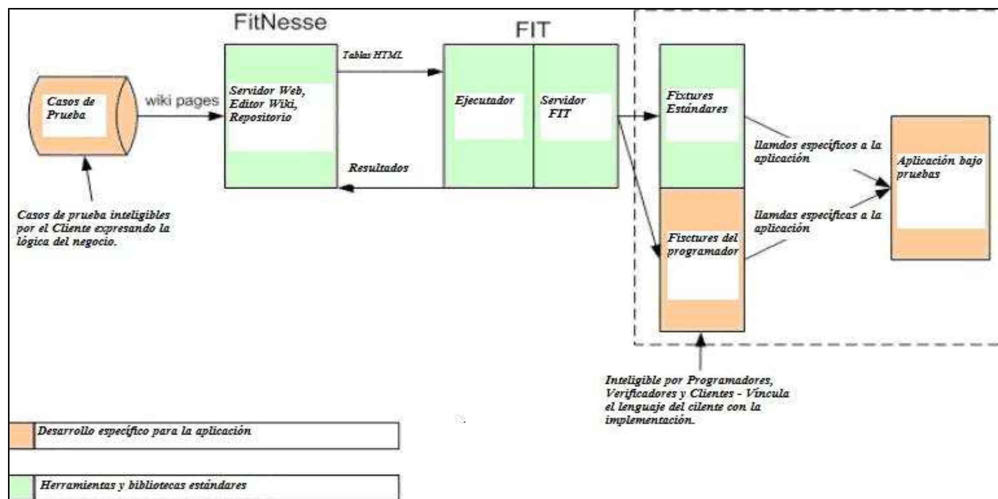


Fig. C- 13: Arquitectura de FitNesse [Aae07].

FitNesse tiene una arquitectura modular por la cual admite ejecutar programas escritos en otros lenguajes diferentes de Java, incorporándole la implementación de FIT y los servidores FIT adecuados, además de las *fixtures* correspondientes.

Para equipos de trabajo la instalación recomendada consiste en que cada desarrollador tenga su propio Fitnesse ejecutando en su equipo local y las pruebas están almacenadas en un servidor centralizado al que se denomina "FitNesse Server". Esta arquitectura conocida como "FitNesse distribuido" se muestra en la figura C-14.

[124] Son el "equivalente" FitNesse a "*web folders*" o *c# namespaces*.

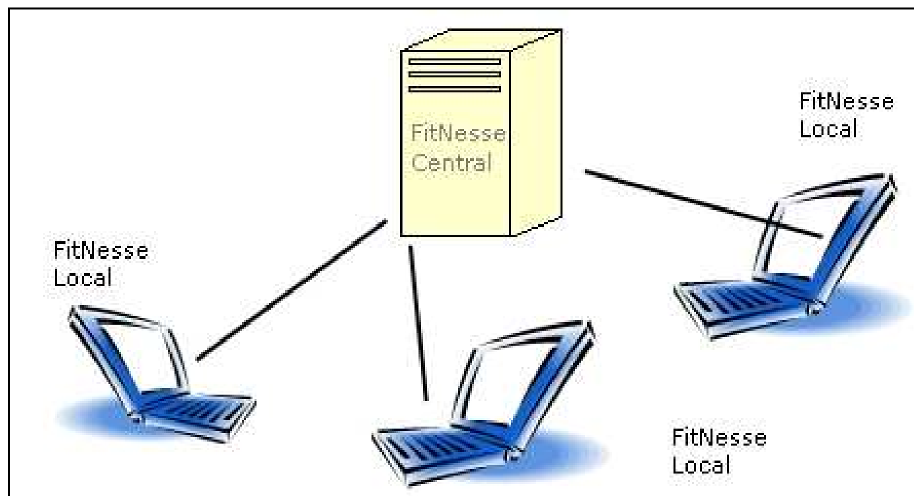


Fig. C- 14: FitNesse Distribuido.

Metodología de trabajo con FitNesse distribuido.

Los desarrolladores copian las pruebas definidas por los usuarios y verificadores desde el servidor, escriben las *fixtures* necesarias, prueban localmente y devuelven al servidor los resultados.

.

C.3 Integración con otras herramientas

FitNesse puede integrarse o vincularse con otras herramientas. En esta sección se tratará en particular la vinculación con la herramienta para prueba de aplicaciones *web* Selenium [Sel06] y se enumerarán otros casos, brindándose referencias.

C.3.1 FitNesse y Selenium

Selenium [Sel06] es una herramienta que permite automatizar las pruebas de aceptación y funcionales de aplicaciones con interfase *web* y validar la compatibilidad de dichas aplicaciones con diferentes navegadores y sistemas operativos. Está basada en JavaScript, se ejecuta desde un navegador real apuntando a un servidor *web* que la tenga instalada y es ampliamente utilizada en su ámbito de aplicación específico. Las pruebas para Selenium se definen en tablas HTML, indicando datos y comandos para proveer opciones de navegación, validar páginas, comprobar la aparición de elementos y otros.

Ofrece una herramienta complementaria, Selenium IDE, que permite grabar en el formato de tablas adecuado mientras se ejecuta una sesión utilizando el navegador FireFox [Moz07]. Dichas tablas así creadas permitirán repetir la prueba pudiendo ser editadas por los desarrolladores. Es posible también comandar su ejecución "programaticamente" mediante las API *Selenium Remote Control (SeleniumRC)* lo cual permite su ejecución remota y sincronizada con otras herramientas.

Las propuestas de vinculación con FitNesse que se mencionarán proponen utilizar a FitNesse para crear, editar y comandar la ejecución de las pruebas bajo Selenium, así como para almacenar los resultados De esta forma se aporta un repositorio común a las pruebas donde organizar y documentar las mismas, publicar los resultados, mantener versiones y permitir que usuarios no técnicos utilicen FitNesse para elaborar las pruebas que serán luego ejecutadas vía Selenium. Utilizan *DoFixture*, lo que también permite implementar un lenguaje específico de dominio (*DSL*) tal que los verificadores y usuarios puedan definir las pruebas de aplicaciones *web* de manera que les resulte legible e inteligible [Mil106]. Entre las propuestas en tal sentido se encuentran [Foy06], [Mil106], [Vla06], [Sti06]. Gojko Adzic en [Adz307] brinda un resumen acerca del mecanismo de integración y provee a tales efectos de la *fixture* especializada "WebTest"[125].

La figura C-15 muestra como luce la integración en una ejecución utilizando las *fixtures* suministradas por Vlagsma [Vla06]. En la figura C-16 se observa la herramienta Stiq[126] [Sti06] durante su ejecución.

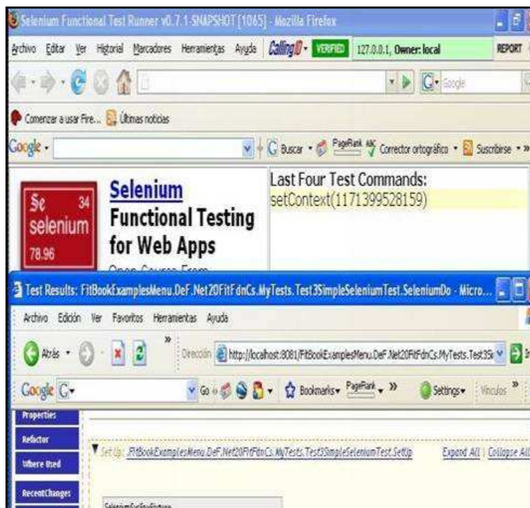


Fig. C- 15: Integración FitNesse Selenium.

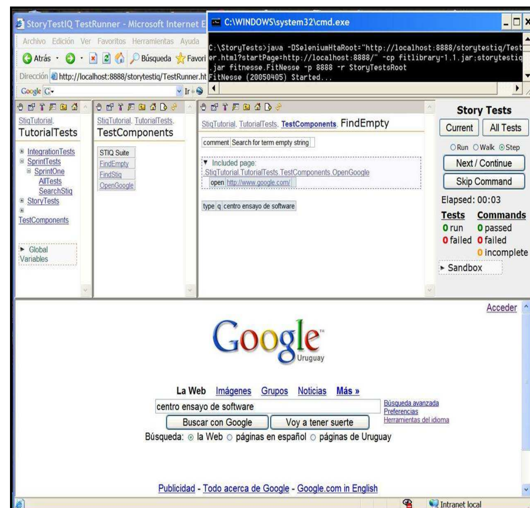


Fig. C- 16: Herramienta Stiq (StoryTestIQ).

En la tabla siguiente se brinda un ejemplo de especificación parcial de una prueba aplicando *DoFixture*, con referencias al código Java de la *fixture* interprete que utiliza las API *Selenium Remote Control (SeleniumRC)*, según Cory Foy [Foy06].

[125] http://gojko.net/fitnesse/webtest/
[126] StoryTestIQ.

```
|! – fitnesse.SeleniumRunner-! |
|Set server to|localhost|and port to|4444|and browser to|*firefox|and domain to|http://www.google.com|
```

```
import fitlibrary.DoFixture;
public class SeleniumRunner extends DoFixture {
        Selenium seleniumInstance = null;
        public SeleniumRunner() {
        }
        public boolean SetServerToAndPortToAndBrowserToAndDomainTo(String
server, int port, String browser, String domain) {
        seleniumInstance = new DefaultSelenium(server, port, browser, domain);
        seleniumInstance.start();
        return true;
    }
```

```
|The user navigates to the url| http://www.google.com|
|The page has the title|Google|
|The page has one element named|g|
|The page has one element named|btnG|
|The user type|Cory Foy|in the field named|g|
|The user clicks on the button named|btnG|
|The user close the browser||
```

```
Public boolean TheUserCloseTheBrowser() {
      seleniumInstance.close();
      return true;
  }
 Public boolean TheUserNavigatesToTheUrl(string url)
       seleniumInstance.open(url)
       return true;
  }
```

Tabla C- 1: Tablas *DoFixture* (FitNesse+Selenium) y clases *C#* interpretes [Foy06].

Otro mecanismo similar de integración ha sido propuesto por Miller [Mil106].

C.3.2 Integración con otras herramientas

A continuación se enumeran otras propuestas de integración:

- Confit: Desarrollado por "BandXI international" [Con06], integra FitNesse en Eclipse.
- FitClipse: El desarrollo fue realizado en la Universidad de Calgary [FIT06] [FIT207] [FIT307]. Se basa en tres componentes: una extensión para Eclipse, un servidor FitNesse y la base de datos MySql[127] . Su objetivo es soportar una metodología basada en EATDD manteniendo un histórico de resultados y reportes que permitan diferenciar las fallas por regresión de aquellas producidas por funcionalidad no implementada [FIT207].

[127] Base de datos *open source* (http://www.mysql.com).

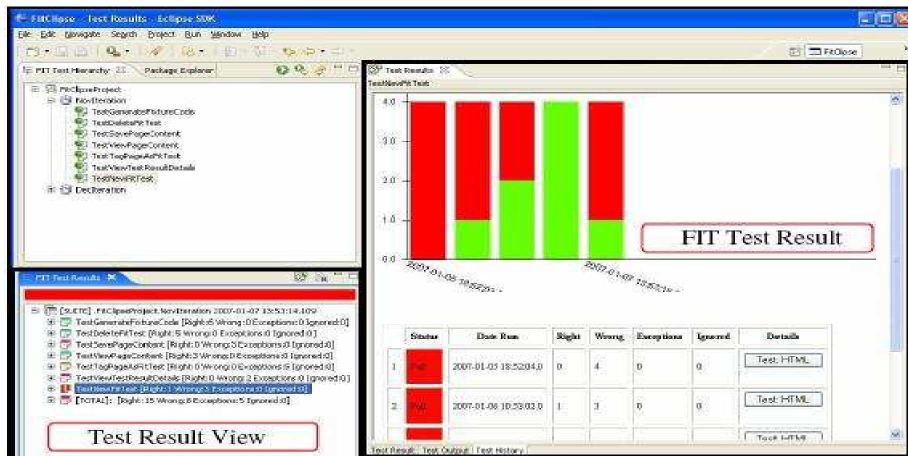


Fig. C- 17: FITClipse [FIT207].

- Ant y Cobertura [Wie06]: Propone una forma de medir el cubrimiento de código utilizando Ant, Cobertura[128] [Cob07] y FitNesse.
- CruiseControl[129] [Jai206]: Propone un mecanismo a los efectos de certificar las construcciones (*builds*) bajo integración continua.

C.4 Consideraciones

FitNesse ofreció una respuesta a las carencias de FIT (5.5), especialmente brindando un repositorio organizado para las pruebas y una interfase de elaboración, colaboración, ejecución y visualización amigable. A partir de información disponible sobre experiencias y evaluaciones empíricas se propondrán las siguientes consideraciones sobre FitNesse. Se destaca que además aplican varias de las consideraciones efectuadas al respecto en el análisis sobre FIT que se brindó en (5.5), en donde se enunciaron las principales debilidades que afectan su viabilidad y la de toda otra herramienta basada en FIT que no las resuelva.

Fortalezas:

- **Consolida las pruebas y su documentación en un *wiki***, facilitando la comunicación y centralizando la documentación y los casos de prueba.
- **Publica el resultado de la ejecución de las pruebas** facilitando el retorno hacia todos los participantes del proyecto.
- **Apoya la implementación de EATDD** [Mel07].
- **Su código es libre.**
- **Brinda la posibilidad de utilizar JavaScript en las páginas.**
- **De rápido aprendizaje e instalación inmediata.**
- **Su arquitectura modular** resulta apta para implementar pruebas en otros lenguajes diferentes de Java e integrarle a otras herramientas.

[128] Herramienta *open source* para efectuar pruebas de cubrimiento de código Java.
[129] Herramienta *open source* para integración (*builds*) continua (http://cruisecontrol.sourceforge.net).

Debilidades:

- **Consolida las pruebas en un *wiki*** no integrado al ambiente de desarrollo.
- **El versionado de las pruebas**, si bien existe, es muy simple. Tampoco se guarda historia acerca de las ejecuciones de las pruebas.
- **No brinda funcionalidades para apoyar la gestión del proceso** de pruebas.

Estas debilidades se han tratado de fortalecer mediante la integración con otras herramientas.

FitNesse generó entusiasmo; detectable a partir del importante volumen de información publicado, de los reportes de experiencias, de la integración con otros productos y de la cantidad de descargas efectuadas. En cuanto a este último dato resulta interesante analizar su evolución, que se grafica en la figura C-18.

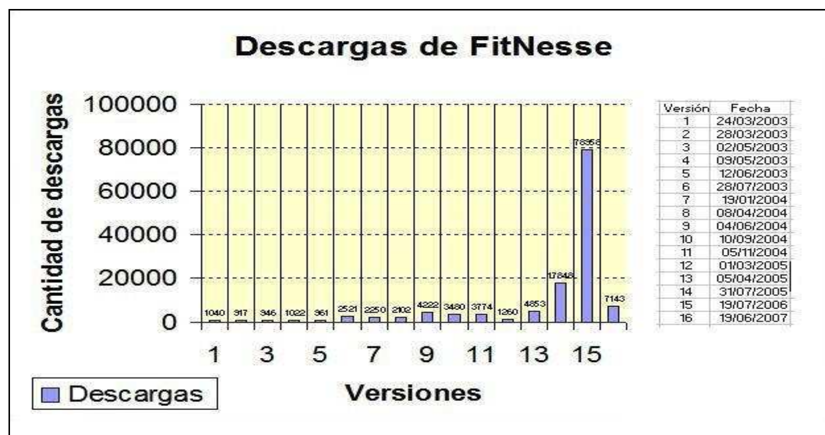


Fig. C- 18: Descargas de FitNesse.

Tal como muestra la figura precedente, las descargas de FitNesse presentaron un pico en el año 2006 habiendo disminuido notoriamente en el 2007: ¿implicó esto una pérdida de interés?, ¿hubo dificultades en su utilización que determinaron esta caída? Probablemente influyeron dos de las debilidades más importantes, coincidentes con las debilidades de FIT no resueltas por la herramienta, a saber: la reconstrucción de los casos de prueba ante cambios en la especificación de las pruebas y del SUT y la cantidad creciente de *fixtures* a mantener. Es posible encontrar en Internet apreciaciones al respecto, por ejemplo Scott Bellware en [Bel07], Bred Pettichord [130]citado por Bellware [Bel06] en sus críticas a FitNesse y Jeremy Miller[131] [Mil07] o James Shore [Sho07] opinando sobre FIT; entre otros.

[130] Bred Pettichord hace hincapié en DSL y promueve la herramienta Watir (http://wtr.rubyforge.org), cuyo proyecto lidera .

[131] Jeremy Miller fue uno de los proponentes de la integración con Selenium y creador de la herramienta Storyteller. A pesar de sus críticas sostiene que se debe seguir intentando y valora la ejemplificación como vehículo de expresión: http://codebetter.com/blogs/scott.bellware/archive/2007/10/28/170346.aspx#comments 29/10/2007.

En particular se entiende necesario investigar para mejorar los aspectos críticos de FIT que se constituyen en sus principales debilidades e impedimentos. En tal sentido es interesante destacar que han elaborado herramientas competitivas con FitNesse que utilizan conceptos de FIT o su código, como ser Autat [SØ05], GeenPepper [GP07], Storyteller [Mil206] y ZiBreve[132] [Mug207].

[132] Herramienta liberada en versión beta por Rick Mugridge, sin disponibilidad sobre el código fuente.