Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Tesis de Licenciatura en Ciencias de la Computación

"***Generación automática de Test de Unidad mediante la captura de Test de Sistema***"

PdH. Victor Braberman, *Director*, y Fernando De Simoni, *Tesista*

***Resumen*—Las características principales de los *Test de Unidad* en un ambiente de desarrollo Orientado a Objetos de cierta *clase* radican en ser focalizados y eficientes.**

La misión de los *Test de Unidad* es es poder verificar el código de una la clase, ejercitando a la *clase* como si ésta lo hiciera con otras *clases* del sistema.

La generación de *Test de Unidad* representativos no es una actividad trivial, sino que requiere de dominio de la especificación de la unidad.

Bajo este contexto, generar los valores de entrada para lograr que el *Test de Unidad* cubra la utilización de la *clase* no es sencillo. Estos *Test de Unidad* pueden ser muy volátiles o ser utilizados con otros valores al integrarse con otras *clases* del *Sistema*.

Los *Test de Unidad* automáticamente generados tienen el potencial de descubrir ejecuciones de sistema intrínsecamente complicadas a bajo costo.

La generación automáticamente de *Test de Unidad* permite reducir el costo de generación y mantenimiento de los mismos. De esta forma, se facilitan técnicas altamente utilizadas como *Extreme Programming* (XP) y algunas de sus prácticas: *Refactoring*, *Test Driven Development*(TDD) y *continuous testing*.

En la presente tesis presentamos una *estrategia* y una *implementación de referencia* integrada al ambiente de desarrollo, para la captura de *Test de Sistema* y a partir de este generar automáticamente *Test de Unidad* y oráculos de estos tests..

***Index Terms*—Test de Unidad, Test Factoring, captura de ejecución de sistema, verificación automática de sistemas críticos, matriz de trazabilidad, Ingeniería de Software.**

I. Introducción

En 1987 Frederick P. Brooks introduce en su artículo "***No silver bullet***" [NSB1986], la llamada crisis del *Software* y las propiedades inherentes del *Software*: complejidad, conformidad, cambio e invisibilidad.

Brooks afirma que la complejidad del *Software* no es accidental sino una **propiedad esencial del mismo** y que está sujeta a un cambio constante. Más aún, el *Software* que ha resultado exitoso es adaptado nuevamente introduciéndose cambios para aceptar nuevos casos.

Un problema que *Brooks* trae a luz, es la invisibilidad del *Software*, y las relaciones internas, que aún con modelos, son difíciles de representar.

Dentro de este contexto el costo de detectar un defecto se incrementa conforme avanza el tiempo desde la introducción del mismo. Por ejemplo el costo de detectar que un módulo no responde correctamente para ciertos *valores de entrada*[1] es oneroso en un ambiente productivo si este no ha sido detectado anteriormente.

No obstante esto, muchos *sistemas* de *misión crítica* que tienen definido como *driver* la calidad, pueden valerse de varias tácticas para enfrentar estos problemas.

Existen hoy día, herramientas que permiten verificar con precisión el comportamiento del *Software* (Por ejemplo *Model Checking*) [SEP2005]. El problema de estas herramientas es el **alto costo computacional**.

Una estrategia más conservadora es realizar distintos procesos de verificación **durante** un proyecto de *Software* para asegurar la calidad. En particular los procesos de *"Software Testing"* son utilizados para identificar la *correctitud*, *completitud*, *seguridad* y *calidad* del *Software* desarrollado [SEP2005].

El objetivo de estos procesos es descubrir errores tempranamente utilizando, por ejemplo, *Test de Unidad* y *Test de Sistema*[2]. Estos intentos por descubrir los errores forman parte de la respuesta de compromiso a la ley de evolución del *Software*: *"... entropy of Software System continually increases unless steps are taken to control..."* (Lehman & Belady) [IPG1972].

Actualmente es común en proyectos de aplicaciones críticas, realizar estos procesos para reducir la entropía del *Software* y controlar así la calidad del mismo.

En particular, están altamente difundidos en la *Industria del Software*, los *Test de Unidad* para validar el correcto comportamiento al momento de desarrollar una unidad o modificar/extender su funcionalidad.

El responsable de la unidad desarrolla el *Test de Unidad* con la finalidad de validar la funcionalidad de la misma (*métodos*, *clases*, *paquetes*) antes de integrar su desarrollo al *sistema*[3].

PhD. Victor Braberman, es Profesor de la Universidad de Buenos Aires, ha dictado diversas materias de Ingeniería de Software en el Departamento de Computación de la Universidad de Buenos Aires. Sus áreas de investigación principales son "Formal Verification" y "Software Analysis". *PhD. Victor Braberman* dirige un grupo de investigación en Ingeniería de Software, DEPENDEX, el cual ha publicado diversos papers en ediciones de importante referato en la Ingeniería de Software. TEL: +54 11 4576–3359, e-mail: vbraber@dc.uba.ar

Fernando De Simoni es Universitario en Ciencias de la Computación. Se desempeña como ayudante de 1ra regular del mismo Departamento de Computación en el área *Ingeniería de Software*. Desarrolla su carrera profesional en empresas multinacionales en la industria de Software. Su ámbito de influencia en estas empresas se concentra en el Diseño y Arquitectura de sistemas de *misión crítica*. En particular, utilizando la tecnología JAVA/J2EE. TEL: +54 11 4 816–3731, fax: + 54 11 4816–3731, e-mail: ldesimon@dc.uba.ar

[1] Utilizaremos valores de entrada refiriéndonos genéricamente a los parámetros o argumentos que reciben los métodos de las *clases*.

[2] En este capítulo se definirán estos conceptos.

[3] Incluso en algunas metodologías [WEBEXT] antes de desarrollar la misma *clase* o unidad.

El *Test de Unidad* se concentra en la funcionalidad individual de la *clase* o componente, sin tener en cuenta a las demás *clases* o componentes del *sistema*. Estos test son útiles para verificar en forma más eficiente la operación de la *clase*, incluso facilitan la depuración o *debug* de la *clase*.

Los *Test de Unidad*, a su vez, son parte vital de ciertos procesos de desarrollo como por ejemplo *Extreme Programming* (XP) [WEBEXT] y algunas de sus prácticas: *Refactoring* [WEBREF], *Test Driven Development*(TDD) [4] y *continuous testing* [STT2005].

La generación de ***Test de Unidad*** representativos no es una actividad trivial, sino que requiere del dominio de la especificación de la *clase*. Muchas veces, ésta especificación es desconocida, en cuyo caso, se pueden generar *Test de Unidad* que no representan el comportamiento completo de la *clase*.

Incluso con una especificación formal de la *clase*, ésta puede evolucionar en su utilización sin modificar el comportamiento interno de la *clase*. Entonces, los *Test de Unidad* deberían representar esta nueva utilización de la unidad.

En el caso de no contar con una especificación formal de la *clase*, una alternativa es analizar la implementación realizando un cubrimiento de la ejecución mediante *Test de Caja Blanca* [TAS2004] o *Testing Estructural* [SEP2005]. Debido al importante conocimiento que se debe tener de la clase se requiere un alto costo para generarlos en forma manual. Sin embargo, la generación completa en forma automática puede generar múltiples *Test Estructurales* poco interesantes [TAS2004].

Al momento de generar los valores de entrada de los *Test de Unidad* es importante poder generar un conjunto de valores representativos y completos del comportamiento de la *clase*. Es decir, comprender e implementar *Test de Unidad* que **cubran** el comportamiento que la *clase* tendrá al ser integrada al *sistema*.

Otro desafío de la generación de *Test de Unidad* es construir un ambiente donde poder ejecutar los *Test de Unidad*. Este desafío usualmente es evitado utilizando *frameworks* de test de unidad como *JUnit* [JUN2001], entre otros [UTF2004].

La realidad indica también, que muchos *sistemas* ya desarrollados carecen de estos *Test de Unidad*, y el esfuerzo para generarlos es importante, dado el conocimiento requerido para esta actividad.

Los ***Tests de Sistema*** son usualmente desarrollados basándose en documentos de alto nivel que describen la funcionalidad del *sistema* desde la perspectiva de los *actores*[5] (Por ejemplo documentos de Requerimientos, Manuales de Usuario u otros). Una de las características de la generación de estos *Test de Sistema*, es que no es indispensable el conocimiento de los procesos internos que realiza el *sistema*, generando la característica de no depender de la implementación interna del *sistema*.

Asimismo la experiencia requerida para generar estos test es menor que la necesaria para generar *Test de Unidad*.

Otra característica importante de los *Test de Sistema* es que pueden descubrir, debido a la interacción de unidades, errores que los *Tests de Unidad* no revelan. *Martin Fowler* incluso sugiere que los defectos detectados por un *Test de Sistema*, deberían generar agregar un *Test de Unidad* que exponga el mismo defecto, para que este defecto no vuelva a ser reingresado.

El **descubrimiento de errores** y reparación mediante *Test de Sistema* demandan un proceso más largo y costoso que los hallados mediante *Test de Unidad* por el propio desarrollador.

Sin duda los *Test de Sistema* son sumamente valiosos y esenciales, pero si éstos no están automatizados, el costo asociado a la generación, preparación del ambiente y ejecución, es **alto** y **vólatil**.

En *sistemas* críticos de mediana dimensión ejecutar estos test puede durar días o semanas y requiere un alto esfuerzo humano [RTI1998]. El esfuerzo aplicado a la ejecución de los *Test de Sistema* puede no cubrir todo el rango de posibilidades de ejecución. Los *Test de Unidad* se pueden complementar con los *Test de Sistema*.

Un ejemplo de esta última afirmación, es el daño irreparable que causo el mal funcionamiento del *Software* de la *Therac-25* [MDT1995] entre 1985 y 1987, administrando a pacientes dosis erroneas de radiación. Luego de un minucioso análisis [MDT1995] se llego a la conclusión, entre otras, que los *Test de Sistema* realizados sobre el *Software* de la *Therac-25* no fueron suficientes y *Test de Unidad* y *Test de Regresión* deberían haber sido utilizados.

De la misma forma, el **primer lanzamiento** del cohete *Ariane 5*, en 1996, tuvo que ser detonado a los 40 segundos de vuelo, con perdidas directas de 1200 Millones de Dolares. Ésto, debido a una excepción al intentar convertir un número de *64 bits* a uno de *16 bits sin signo*. De las conclusiones del comité investigador [A5F1996] surge que una simulación del despegue, aplicada a sus componentes en forma aislada, hubiera detectado la falla.

A. Objetivo y estructura

A partir de esta situación presentada, el objetivo de esta *tesis* es desarrollar una estrategia para generar *Test de Unidad* de *clases* en forma automática, mediante captura de ejecuciones de *Test de Sistema* del **S**istema **B**ajo **A**náli-

[4] Test-Driven Development (TDD) [TDD2005] es una metodología que utiliza *tests* para ayudar a los desarrolladores a tomar las decisiones correctas en el momento apropiado. TDD no es *testing*, sino la utilización de estos test para crear *Software* de una manera sencilla e incremental.

[5] Definimos un **actor** como el rol que asume una persona o sistema que interactúa con el *sistema* que estamos construyendo. Tiene la propiedad de ser externo al *sistema*. Hay que tener en cuenta que un usuario puede acceder al *sistema* como distintos *actores*.

sis SUA [6].

Se ha desarrollado una *prueba de concepto* con una *suite* de herramientas novedosas integrada al ambiente de desarrollo del desarrollador para brindarle mayor control al momento de modificar una unidad del *sistema*. Asimismo hemos generado un ambiente que permita la ejecución de estos test para validar el correcto funcionamiento de la unidad. Estos *Test de Unidad*, automáticamente generados, se podrían agregar a los ya existentes para evaluar a la unidad en situaciones más complejas y realistas, logrando una mayor cobertura y duración de los *Test de Unidad*.

Las contribuciones de esta *tesis* incluyen: (i) presentar un ambiente para generar automáticamente *Test de Unidad* a partir de *Test de Sistema* y ejecutar estos test generados,(ii) analizar e implementar la estrategia novedosa correcta para almacenar suficiente información de estos *Test de Unidad* (iii) analizar los desafíos de esta estrategia y plantear la forma de solucionarlos.

En los próximos capítulos presentaremos ***nuestra propuesta*** (Capítulo III), desarrollando un modelo computacional simplificado con las *definiciones operacionales*[7] necesarias. Presentaremos también una Comparación con otras propuestas del *Estado de la Práctica* y Justificación de la propuesta (Sub-capítulo III.D)

Luego introduciremos ***nuestra estrategia***, (Capítulo IV), es decir la metodología [8] **que implementa** *nuestra propuesta*.

En base a esta estrategia, presentaremos una ***Implementación práctica***, (Capítulo V), de ***nuestra estrategia*** con herramientas desarrolladas en *JAVA* [WEBJAV] para un ambiente de desarrollo. Bajo este contexto de implementación, ejemplificaremos un caso de utilización simple (Sub-capítulo V.C) de las herramientas y un proceso definido para su utilización (Sub-capítulo V.D).

Por último presentaremos los **Futuros Trabajos**, (Capítulo VI), y **Conclusiones** (Capítulo VII) que nuestra *tesis* ha identificado.

[6] SUA: System Under Analisys.

[7] Las **definiciones operacionales** son definiciones limitadas, cuyo propósito es permitir que el lector de esta tesis se acerque a los aspectos de la realidad a estudiar con nuestro marco referencial.

[8] Entendemos por **metodología** la etapa específica que continúa una posición teórica y epistemológica. Esta da pie a la selección de técnicas concretas de investigación.

II. Test de Unidad

(The Programmer) personally defines the functional and performance specifications, designs the program, codes it, test it, and writes its documentation. They need great talent, ten years experience and considerable systems and applications knowledge, whether in applied mathematics, business data handling, or whatever!

–Fred P. Brooks

El objetivo de los *Test de Unidad* es asegurar la confiabilidad de las unidades del *sistema* que se producen para cumplir con cierto requerimiento. El proceso intenta explicitar contratos que las unidades deberán cumplir. Los *Test de Unidad* son realizados por el equipo de desarrollo bajo la supervisión del líder de este equipo.
Usualmente el desarrollador que "codifica" el *sistema* es quien diseña e implementa los *Test de Unidad*. Luego ejecuta éstos para verificar que la unidad de código cumple con los requerimientos para los cuales ha sido creada.

Cada *Test de Unidad* debe ser ejecutado individualmente y en forma aislada, tomando sus valores de entrada y observando el *resultado obtenido*.

Luego de haber superado exitosamente estos *Test de Unidad*, la unidad debe ser verificada en su comportamiento con otras unidades, es decir la integración.

Los *Test de Unidad* se concentran en cada unidad del *Software* y se basan principalmente en la lógica interna de procesamiento y estructura de datos siempre con una visión limitada a la unidad.

Algunas de las áreas típicamente cubiertas por los test de unidad, son [SEP2005]:

- Incorrecta inicialización de estructuras o incorrecta representación de valores.
- Verificación de cálculos/manipulación realizados por la unidad.
- Verificación de bordes (por ejemplo cuando el máximo o mínimo valor posible es utilizado) .
- Aislar la unidad y verificar el correcto comportamiento individual de la unidad.
- Verificación de liberación de memoria o baja *performance* de las estructuras en múltiples invocaciones.
- Verificación de cumplimiento de aserciones durante la ejecución de cierto método.

En este contexto la definición de "unidad" dependerá de la estrategia de diseño del *sistema* a verificar y de la tecnología seleccionada para el desarrollo.

Por ejemplo en un ambiente de programación orientada a objetos, una unidad puede ser representada por un *class*, o una *instancia* de un *class*, o la funcionalidad implementada por un método.

Existen distintas alternativas para desarrollar *Test de Unidad*, pero entre ellas podemos encontrar los siguientes denominadores comunes:

- Identificar el estado de un *sistema*, el cual se utilizará para inicializar el test.

- Establecer o recrear ese estado inicial.
- Ejecutar los *Test de Unidad* desde este estado inicial.
- Juzgar el *resultado obtenido*.

El alcance de los *Test de Unidad* por definición se concentra en el unidad, por lo tanto no pretende capturar todos los errores del *sistema*. Más aún, los *Test de Unidad* no se concentran en la integración de la unidad al *sistema*, por lo tanto no pretenden detectar errores de rendimiento u otros aspectos globales del *sistema*.

En particular no siempre es posible implementar *Test de Unidad* efectivos que anticipen los valores que recibirá la unidad bajo análisis en la realidad, es por ello que los *Test de Unidad* son efectivos solo al complementarlos con otras actividades de *testing* [SEP2005].

III. Nuestra Propuesta

> *"You can allways write more tests. However, you will quickly find that only a fraction of the tests you can imagine are actually usefull."* –**Test infected: Programmers Love Writing Tests from Beck and Gamma**

En el siguiente capítulo introduciremos *nuestra propuesta*. Para lo cual, será necesario formalizar un modelo computacional simplificado con su notación, que representa la realidad sobre la cual desarrollamos *nuestra propuesta*. Presentaremos, en un alto nivel, la captura de casos y generación de *Test de Unidad* con los desafíos inherentes a resolver.

A. Propuesta

El objetivo de esta *tesis* es desarrollar una estrategia para generar *Test de Unidad* de *clases* automáticamente utilizando para ello ejecuciones de *sistema* provenientes de *Test de Sistema*.

El contexto ideal de *nuestra propuesta* es donde no se han explicitado los contratos de las *clases* mediante *Test de Unidad* o estos contratos no son representativos de la utilización de la *clase*. Cuando es impracticable generar *Test de Unidad* por el conocimiento requerido, o las estructuras complejas que requieren los métodos de las *clases* como párametros y generación de contexto, o inicialización.

Entonces, nos concentramos en tomar ventaja del **esfuerzo** puesto en la etapa de **Testing** y diseño **QA**[9]. Bajo estas etapas se ejecuta la funcionalidad desarrollada y esta experiencia es **volátil**. Para aprovechar este esfuerzo, incorporamos una **base de test de sistema** con los *resultados obtenidos* internamente y los contratos que se han "instanciado", para luego poder generar *Test de Unidad* **funcionalmente válidos**.

Hemos generado una *prueba de concepto* con una *suite* de herramientas novedosas integrada al ambiente de desarrollo del desarrollador para brindarle mayor control **al momento de integrar cambios** sobre una cierta *clase* del *sistema*.

En la *prueba de concepto* de la estrategia, hemos generado una herramienta para adquirir información relevante a esta integración o modificación de *clases*. Luego, presentar esta información relevante al desarrollador; Y automáticamente, verificar el impacto de su cambio sobre una *clase* realizando *Test de Unidad* derivados de *Test de Sistema* almacenados, exitosos, que tienen injerencia en la modificación introducida.

Nuestra hipótesis se basa en el beneficio de ejecutar *Test de Unidad* representativos a la modificación, en lugar de test generales.

No se han realizado al momento experimentos de envergadura que demuestren empíricamente el beneficio de esta propuesta. El alcance de esta tesis se concentra en la factibilidad de la propuesta y en el beneficio asociado a la misma (Ver Sub-capítulo III.D).

[9] *Quality Assurance* [SEP2005]: asegura la existencia y efectividad de los procedimientos que se realizan para verificar que los niveles de calidad serán logrados.

B. Modelo Computacional

A los efectos de formalizar *nuestra propuesta* definiremos un modelo de representación de un *sistema* desarrollado bajo el **paradigma de objetos** [SEP2005], sus estados y la ejecución de este *sistema*.

Un **sistema** [OOS2004] es un conjunto organizado de partes que se comunican para lograr resolver objetivos especificos, para los cuales han sido desarrollados. Los *sistemas* pueden ser descompuestos en subdivisiones, hasta llegar al nivel más básico de nuestro *modelo computacional*: las *clases*, que modelan objetos de la vida real.

Una **clase** [OOS2004], es una abstracción de las unidades de la realidad, del modelo de representación orientado a objetos. Esta unidad se formaliza como las definiciones de las propiedades y comportamiento concreto.

Un **objeto** [OOS2004] o **instancia** de una *clase* es una entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos) en **tiempo de ejecución** .

La **instanciación** [OOS2004] de una *clase*, es la lectura de las definiciones de la *clase* y la creación de una **instancia** de la *clase*[10].

Bajo nuestro modelo un **método** [OOS2004] de una *clase* es un algoritmo que puede ser ejecutado por una **instancia**, cuya ejecución se desencadena tras la recepción de un **mensaje**. Desde el punto de vista del comportamiento, es lo que la *instancia* puede realizar. Un método puede producir un cambio en las propiedades de la *instancia*, o la generación de uno o más mensajes con otras *instancias* del *sistema*.

Los **eventos** [OOS2004] son abstracciones que representan un tipo de estimulacion para la cual el *sistema* posee una respuesta estandarizada. En el sub-capítulo IV.A.1 presentaremos una definición metodológica más formal.

Los nombres de las **clases**, $\mathcal{C}$, serán únicos, notaremos a las *instancias* de las *clases* con la letra minúscula, *c*. Los **métodos** se notarán $\mathcal{M}$ y estos definen unívocamente a la *clase* a la cual que pertenecen $\mathcal{C}$. Los **parámetros** de los métodos, $\mathcal{P}$, y las **variables locales**, $\mathcal{L}$, serán ambos relativos a los nombres del método asociado, para hacerlos únicos por composición; Luego, los métodos se componen de parámetros y variables locales. Las **variables locales** $\mathcal{L}$ serán **valores primitivos** [SEP2005] $\mathcal{V}$ o **referencias** $\mathcal{R}$ a otras estructuras (*instancias*).

[10] Bajo esta definición una **instancia** es sinónimo de objeto dado que en definitiva ambos términos refieren al concepto de ser una representación de una *clase*.

B.1 Acción

Definiremos una **acción**, $\boldsymbol{a}$, como la **invocación** o **ejecución** de cierto método $\mathcal{M}$ [11] de cierta *instancia* $\boldsymbol{c}$ y lo notaremos como $\overset{\mathcal{M}(\mathcal{P}_1,...,\mathcal{P}_n)/Pos}{\rightarrow}$, donde el símbolo $\mathcal{M}$ identifica al método invocado,$P_1, ..., P_n$ representan los parámetros o valores de entrada de la invocación con los mismos tipos de Objetos y valores de la ejecución. $\boldsymbol{Pos}$ es definido como $Pos \mapsto \mathcal{V} \cup \mathcal{R} \cup \uparrow$ [12] el valor retornado del mismo tipo de Objeto de la acción.

Proyectaremos el método asociado a la acción mediante la relación $\overset{metodo}{\rightarrow}$, por ejemplo dada cierta acción $\boldsymbol{a}$ notaremos la proyección de su método: $\boldsymbol{a} \overset{metodo}{\rightarrow} \mathcal{M}$ donde $\mathcal{M}$ es el método asociado a la acción. De la misma forma proyectaremos los parámetros de entrada de la acción mediante $\overset{parametros}{\rightarrow}$. Por último el resultado de la invocación *Pos* se proyectará utilizando $\overset{pos}{\rightarrow}$. Como a partir de cierto método $\mathcal{M}$ es conocida la *clase* $\mathcal{C}$, entonces dada cierta acción $\boldsymbol{a}$, $\boldsymbol{a} \overset{clase}{\rightarrow} \mathcal{C}$.

Dado que la acción se realiza en cierto contexto de ejecución, proyectaremos con $\overset{instancia}{\rightarrow}$ la **instancia** especifica sobre la cual se ha ejecutado la acción. Las *instancias* poseen un identificador $\boldsymbol{id}$, a los efectos de diferenciar dos *instancias* de la misma *clase*. Este identificador $\boldsymbol{id}$ es único para toda *instancia* dentro la ejecución de un *sistema*.

Tanto los parámetros, $\mathcal{P}_1, ..., \mathcal{P}_n$,como el resultado, *Pos*, de cada acción se conservan almacenados en cada acción, por ello diremos que cada acción es **autocontenida**.

B.2 Secuencia de Acciones

Luego, una secuencia de acciones $\bar{\sigma}$, se formaliza como: $\bar{\sigma} = a_1, a_2, ..., a_n$, donde n es un número natural, $n > 0$ y $length(\bar{\sigma})$=n. Las acciones de la secuencia se asumen distinguibles entre si por su precedencia en la secuencia.

B.3 Prefijo de Secuencia de Acciones

Dada cierta secuencia $\bar{\sigma}$, donde $\bar{\sigma} = a_1, a_2, ..., a_n$. Un prefijo de secuencia de acciones, $\bar{\sigma}(r)$, se define utilizando un cierto número natural r, $r > 0$ y $r \leq length(\bar{\sigma})$, donde $\bar{\sigma}(r) = a_1, a_2, ..., a_r$ y $length(\bar{\sigma}(r))$=r.

B.4 Secuencia Restringida de Acciones

Bajo cierta necesidad, podemos filtrar una secuencia de acciones, $\bar{\sigma}$, solo a cierta *instancia* de una *clase* del conjunto de *clases* del *sistema*.

Esta *instancia*, restringe las posibles acciones que son miembro de la secuencia filtrada. En particular, dada cierta *instancia* $\boldsymbol{c}$ de la *clase* $\mathcal{C}$, definimos $\bar{\sigma}/\boldsymbol{c}$ como una secuencia de acciones tal que para toda acción a_i que pertenece a la secuencia restringida, a_i es parte de $\bar{\sigma}$ y $a_i \overset{instancia}{\rightarrow} \boldsymbol{c}$, donde $\boldsymbol{c}$ es *instancia* de la *clase* $\mathcal{C}$.

[11] Implícitamente en el método se encuentra definida unívocamente la *clase* $\mathcal{C}$ a la cual pertenece el método.

[12] Nótese aquí que el valor retornado *Pos* puede ser no definido, $\uparrow$, en caso que la acción no retorne valor alguno dado que el método asociado a la acción no tiene valor de salida en su signatura (void).

C. Formalización de Ejecución de un Sistema

La ejecución de un *sistema* bajo el **paradigma de objetos**, puede formalizarse utilizando una **secuencia de acciones que causan cambios de estados**. A continuación presentaremos una formalización suficiente para desarrollar *nuestra propuesta*.

C.1 Test de Sistema y de Unidad

> *Program testing can be used to show the presence of bugs, but never to show their absence!. –Edsger Dijkstra*

Notaremos st_x a los *Test de Sistema* [13] de cierto *sistema* $\mathcal{X}$.

Cada uno de estos st_x es definido formalmente como una secuencia de ejecución $\bar{\sigma}$. Definiremos también a los *test de unidad* **derivados automáticamente** de este mismo *sistema x* como ut_x. Cada uno de estos ut_x es una **subsecuencia** $\bar{\sigma}$ restringida a cierta *instancia*, de alguna de las secuencias de los st_x.

D. Formalización y Captura de Casos

D.1 Captura de casos

Supongamos entonces cierta ejecución, la captura consiste en la secuencia de acciones de las *instancias* involucradas en el st_x [14] durante la ejecución del st_x [15].

Nuestra propuesta conserva una **base de test de sistema** con la secuencia de acciones sobre las *instancias* involucradas, incluidos los **parámetros completos** y **valor de retorno**, si lo hubiese, de cada una de estas acciones.

Luego de la ejecución se almacena el st_x en la **base de test de sistema**.

Debido a que al generar los *Test de Unidad* sólo ejecutaremos ciertas acciones, es imprescindible que estas acciones sean **autocontenidas** y puedan ser ejecutadas sin ninguna dependencia con acciones ejecutadas anteriormente. Por ello conservamos con la acción los **parámetros completos** (*instancias* de clases $\mathcal{R}$ o valores primitivos $\mathcal{V}$).

Otra alternativa para recrear el estado sería capturar el **estado completo**, es decir el valor de todas las variables de la *instancia*, antes de ejecutar cada acción. Bajo esta otra estrategia no sería necesario recrear el estado *reestimulando* la *instancia*, sino que solo se debería recuperar el estado almacenado. Debido a que es necesario conservar el estado de cada *instancia* antes de realizar cualquier operación consideramos esta inversión onerosa y poco eficiente. En el sub-capítulo C.1 del capítulo IV se presentará un análisis más completo de otras estrategias más detallado.

[13] desarrollado en el cápitulo I.

[14] Se entiende por *instancia* involucrada como toda aquella *instancia* de una *clase* del *sistema* que ha sido estimulada mediante la ejecución de algún método.

[15] Como hemos definido anteriormente una acción involucra la ejecución de cierto método $\mathcal{M}$. de cierta *instancia* $\mathcal{C}$. Diferenciaremos acción de método teniendo en cuenta que la acción es el estimulo que recibe la *instancia* de la *clase* en tiempo de ejecución, mientras que el método es la signatura estática y cuerpo de código de la *clase* que se ejecuta.

D.2 Ejemplo de captura de st_x

A continuación se presentará un ejemplo de captura de secuencia de acciones en base al *diagrama de secuencia* [SEP2005] que se presenta en la figura 1.

Entonces supongamos cierto *sistema* $\mathcal{X}$, con *clases* $\mathcal{C}_0$, $\mathcal{C}_1$ y $\mathcal{C}_2$. Se comienza una ejecución del *sistema*, se genera el *evento* que crea una *instancia* c_0 de la *clase* $\mathcal{C}_0$, en particular se ejecuta el método $\mathcal{M}_0$, por lo que se captura la acción a_0 con los parámetros $\mathcal{P}$.

El algoritmo del método, $\mathcal{M}_0$, es quien crea una *instancia*, c_1,de la *clase* $\mathcal{C}_1$ para ejecutar el método $\mathcal{M}_1$. Se captura entonces la acción a_1, almacenándose los parámetros $\mathcal{P}_1$ y el valor de retorno de la ejecución del método $\mathcal{R}_1$.

Luego es la *instancia*, c_0, de la *clase* $\mathcal{C}_0$ quien invoca al método $\mathcal{M}_2$ de la *instancia*, c_2, de la *clase* $\mathcal{C}_2$. Se captura la acción a_2, almancenándose los parámetros $\mathcal{P}$ y el valor de retorno de la ejecución del método $\mathcal{R}$.

Nuevamente la *instancia* c_0 ejecuta un método sobre la *instancia* creada anteriormente de c_1. Esta vez se invoca el método $\mathcal{M}_3$ con los parámetros $\mathcal{P}_2$ y el valor de retorno $\mathcal{R}_2$, se captura la acción a_3.

La *instancia* c_0 ejecuta esta vez el método $\mathcal{M}_4$ con los parámetros $\mathcal{P}_3$ y el valor de retorno $\mathcal{R}_3$. Por lo tanto se captura la acción a_4.

Por último, se ejecuta el método $\mathcal{M}_5$ con los parámetros $\mathcal{P}$ y el valor de retorno $\mathcal{R}$ capturándose la acción a_5. El *sistema* finaliza la ejecución.

Entonces *nuestra propuesta* almacena en la **base de test de sistema** el st_2 con la secuencia asociada $\bar{\sigma} = a_0\ a_1\ a_2\ a_3\ a_4\ a_5$.

En la figura 1 se presenta la situación desarrollada, con las acciones capturadas.

E. Generación de Test de Unidad

Dada cierta *clase* $\mathcal{C}$, perteneciente al *sistema* $\mathcal{X}$, sobre la cual se ha modificado el cuerpo de cierto método $\mathcal{M}$. *Nuestra propuesta* recrea los ut_x generados automáticamente de los *Test de Sistema* st_x almacenados en la **base de test de sistema**, tal que la secuencia almacenada $\bar{\sigma}$ **contiene** en su secuencia de ejecución almenos una acción a_i, i $\leq$ *length*$(\bar{\sigma})$, tal que a_i $\vec{metodo}$ $\mathcal{M}$.

El *Test de Unidad* ut_x generado tiene injerencia en la modificación, dado que ejecutará el cuerpo del método $\mathcal{M}$ modificado. Llamaremos a esta acción, a_i, que ejecuta el cuerpo del método $\mathcal{M}$, **A**cción **B**ajo **A**nálisis del st_x o simplemente **ABA**.

El ut_x generado a partir st_x *simula* el desempeño del método $\mathcal{M}$ bajo la secuencia de acciones del st_x previamente capturado, es decir, *simula* la ejecución controlada del *sistema*, que se ha ejecutado alguna vez y ha sido almacenado en la **base de test de sistema** (Ver sub-capítulo

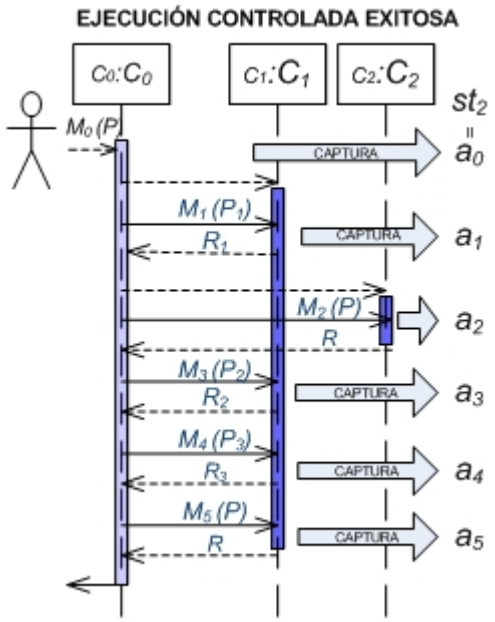

Figura 1. Captura de acciones en ejecución controlada.

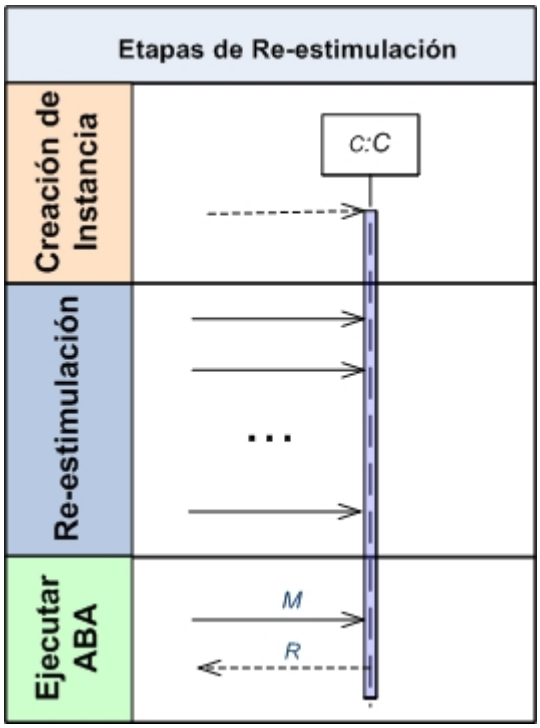

Figura 2. Etapas de Re-estimulación de una instancia

Formalización y Captura de casos D).

Formalizando este concepto diremos que cierto ut_x **es generado** a partir de un st_x mediante el símbolo $\longmapsto$. Entonces, si los ut_1 y ut_2 fueron generados a partir del st_x st_1, podemos afirmar $st_1 \longmapsto ut_1$ y $st_1 \longmapsto ut_2$.

La generación de cada ut_x a partir de un conjunto de st_x (**base de test de sistema**) se realiza mediante los siguientes pasos:

- Selección de los st_x con injerencia al cambio.
- Para cada **ABA** del st_x seleccionado:
  - Crear una nueva *instancia* $\boldsymbol{c}$ de $\mathcal{C}$, donde **ABA** $\vec{clase}$ $\mathcal{C}$.
  - Re-estimulación de la *instancia* $\boldsymbol{c}$ creada filtrando ciertas acciones del st_x.
  - Ejecutar la acción **ABA** sobre la *instancia* $\boldsymbol{c}$ *re-estimulada.*
  - Comparar el **resultado obtenido** con el **resultado almacenado**.

A continuación desarrollaremos cada uno los pasos.

E.1 Selección de los st_x con injerencia al cambio

En el sub-capítulo D, hemos formalizado la captura de casos y de que forma nuestra **base de test de sistema** incorpora los st_x.

Dada cierta *clase* $\mathcal{C}$ y cierto método $\mathcal{M}$ bajo análisis, *nuestra propuesta* selecciona todos aquellos st_x, tal que en la secuencia de ejecución asociada, $\bar{\sigma}$, exista al menos una acción a_i, y a_i $\vec{metodo}$ $\mathcal{M}$, a_i $\vec{clase}$ $\mathcal{C}$.

Todos los st_x que cumplan con lo antes mencionado pueden potencialmente generar un ut_x, es decir $st_x \longmapsto ut_x$.

En el caso de que no existan st_x que cumpla lo antes mencionado no será posible generar un ut_x dado que en la **base de test de sistema** no se han registrado acciones con injerencia al cambio introducido en el método $\mathcal{M}$.

En particular en cierto st_x -con su $\bar{\sigma}$ asociado- puede existir cierta acción $a_i/a_i \in \bar{\sigma}$ y otra acción $a_j/a_j \in \bar{\sigma}$. Donde $i \neq j$ tal que $a_i \in \bar{\sigma}$, a_i $\vec{metodo}$ $\mathcal{M}$ y a_i $\vec{clase}$ $\mathcal{C}$ y $a_j \in \bar{\sigma}$, a_j $\vec{metodo}$ $\mathcal{M}$ y a_j $\vec{clase}$ $\mathcal{C}$. En este caso, se podrán generar dos ut_x, en el primero de ellos la acción **ABA** será a_i y en el segundo la acción **ABA** será a_j.

E.2 Crear una nueva *instancia* $\boldsymbol{c}$ de la *clase* y Re-estimulación

Dado cierto st_x, su $\bar{\sigma}$ asociado e identificado previamente la acción **ABA**, a_i , *nuestra propuesta* **crea** y **re-estimula** una nueva *instancia*, $\boldsymbol{c}$, de la *clase* $\mathcal{C}$ utilizando parte de la secuencia $\bar{\sigma}$.

Es decir **creamos una nueva instancia** $\boldsymbol{c}$ de la *clase* $\mathcal{C}$ (donde $\boldsymbol{c}$ es una *instancia* del tipo de la *clase* asociada a la acción a_i) y luego se ***re-estimula*** la *instancia* aplicando acción a acción sobre esta, **ciertas** acciones de la secuencia $\bar{\sigma}$ del st_x.

Una acción a_j de $\bar{\sigma}$ se ejecutará sobre la *instancia* creada **si y solo si** a_j $\vec{instancia}$ $\boldsymbol{c}$ y a_i $\vec{instancia}$ $\boldsymbol{c}$. Es decir, se ejecuta la acción a_j si y solo si la acción a_j se había ejecutado sobre la misma *instancia* de a_i o dicho de otra forma coinciden los ***id*** de las *instancias* de ambas acciones a_j y a_i.

Para minimizar la creación de objetos, la ejecución del *Test de Unidad* se realiza *re-estimulado* acción a acción **hasta** la acción inmediata anterior a la acción **ABA** de la **secuencia restringida** a la *instancia* creada, $\boldsymbol{c}$, de la *clase* $\mathcal{C}$ bajo análisis. Es decir $\bar{\sigma}(i)/\boldsymbol{c}$, donde i es la posición de a_i en la secuencia de st_x. El resto de las acciones de la secuencia no revisten interés dado que es en la acción **ABA** donde se realiza la ejecución del método $\mathcal{M}$.

Esta ejecución acción a acción, supone que la serialización de parámetros es profunda, es decir que todo objeto que puede ser referenciado en cada método es alcanzado y es posible operar sobre el. Esto gracias a que los parámetros de las acciones son **autocontenidos**.

La ejecución de cada acción se realiza estimulado el método de la acción sobre la *instancia* creada, $\boldsymbol{c}$, de la *clase* $\mathcal{C}$ con los parámetros que cada acción poseé en $\bar{\sigma}$. Cada una de estas acciones ejecutadas modifica, potencialmente, el estado de la *instancia* $\boldsymbol{c}$ de la *clase* $\mathcal{C}$.

E.3 Ejecutar la acción **ABA** y Comparar el *resultado obtenido* con el *resultado almacenado*

Luego de haber logrado *re-estimular* la *instancia*, $\boldsymbol{c}$, creada de la *clase* $\mathcal{C}$, se procede a invocar a la acción **ABA** sobre la *instancia re-estimulada* para obtener el resultado de la acción y luego realizar la comparación del **resultado obtenido** con el **resultado almacenado** en $\bar{\sigma}$. En caso de que la ejecución de la acción **ABA** no retorne resultado alguno, se podría verificar la equivalencia de estados alcanzados, sin embargo esta alternativa no es cubierta por *nuestra propuesta*, sino que se enmarca dentro de Trabajos Futuros (Ver Capítulo VI).

En caso de obtener **resultados equivalentes**[16], se puede afirmar que el cambio introducido en el método $\mathcal{M}$ de la unidad $\mathcal{C}$ inicialmente no ha generado efectos colaterales sobre los contratos explícitos que se dispone y debe cumplir la clase. Sin embargo, si los resultados difieren se deben realizar verificaciones o análisis ulteriores para asegurar que el comportamiento de la unidad responde a las expectativas del desarrollador que ha introducido el cambio.

Existen varias estrategias para analizar las diferencias encontradas al generar algún ut_x. Por ejemplo es posible ejecutar $\bar{\sigma}/\boldsymbol{c}$ sin modificación y $\bar{\sigma}/\boldsymbol{c}$ con la modificación del método al mismo tiempo, acción a acción, para establecer tempranamente las diferencias. Esta estrategia y otras serán analizadas en el sub-capítulo B.1 del capítulo IV.

E.4 Ejemplo de generación de ut_x

Por ejemplo, supongamos cierto *sistema* $\mathcal{X}$, con una **base de test de sistema** que incluye al st_2 (En el subcapítulo D.2 se ha desarrollado la creación de st_2) con:

[16] Se desarrollará el concepto de equivalentes en el sub-capítulo B.1 del capítulo IV.

$\bar{\sigma} = a_0\ a_1\ a_2\ a_3\ a_4\ a_5$. Donde
$a_0\ \vec{instancia}\ c_0$, $a_0\ \vec{clase}\ \mathcal{C}_0$ y $a_0\ \vec{metodo}\ \mathcal{M}_0$.
$a_1\ \vec{instancia}\ c_1$, $a_1\ \vec{clase}\ \mathcal{C}_1$ y $a_1\ \vec{metodo}\ \mathcal{M}_1$.
$a_2\ \vec{instancia}\ c_2$, $a_2\ \vec{clase}\ \mathcal{C}_2$ y $a_2\ \vec{metodo}\ \mathcal{M}_2$.
$a_3\ \vec{instancia}\ c_1$, $a_3\ \vec{clase}\ \mathcal{C}_1$ y $a_3\ \vec{metodo}\ \mathcal{M}_3$.
$a_4\ \vec{instancia}\ c_1$, $a_4\ \vec{clase}\ \mathcal{C}_1$ y $a_4\ \vec{metodo}\ \mathcal{M}_4$.
$a_5\ \vec{instancia}\ c_1$, $a_5\ \vec{clase}\ \mathcal{C}_1$ y $a_5\ \vec{metodo}\ \mathcal{M}_5$.

Supongamos que se ha realizado una modificación en el método $\mathcal{M}_4$ de la *clase* $\mathcal{C}_1$. Entonces el st_x es seleccionado para generar un ut_x y la **ABA** es a_4 debido a que $a_4\ \vec{clase}\ \mathcal{C}_1$ y $a_2\ \vec{metodo}\ \mathcal{M}_4$.

Para generar la subsecuencia ut_x se toman las acciones hasta la acción **ABA**.
Entonces:
$\bar{\sigma}(4) = a_0\ a_1\ a_2\ a_3\ a_4$.

Luego es necesario filtrar, sólo las acciones que se ejecutan sobre la *instancia* c_1 de la *clase* $\mathcal{C}_1$, dado que $a_4\ \vec{instancia}\ c_1$, entonces:
$\bar{\sigma}(4)/c_1 = a_1\ a_3\ a_4$.

Nótese que las acciónes a_0 y a_2 han sido filtradas, dado que $a_0\ \vec{instancia}\ c_0$ y $a_2\ \vec{instancia}\ c_2$, es decir a_0 y a_2 son acciones sobre otras *instancias* de otras *clases*.

La *re-estimulación* de la *instancia* c_1, comienza por la creación de una *instancia* de la *clase* c_1 y luego *nuestra propuesta* genera los eventos para *re-estimular* a esta *instancia* con las acciones a_1 y a_3.

Para a_1 y a_3 se invocan a los métodos asociados de la *instancia* con los parámetros **almacenados** en las acciones, es decir primero se ejecuta $\mathcal{M}_1$ con parámetros $\mathcal{P}_1$ donde $a_1\ \vec{parametros}\ \mathcal{P}_1$.

Luego se ejecuta la acción a_3, entonces se invoca al método $\mathcal{M}_3$ con los parámetros $\mathcal{P}_2$, donde $a_3\ \vec{parametros}\ \mathcal{P}_2$.

Hasta aquí se ha *re-estimulado* a la *instancia* creada que representa a la *clase* $\mathcal{C}_1$. Es ahora, donde se ejecuta la ***ABA***, que es nuestro caso de estudio dado que ejecuta al **método modificado** $\mathcal{M}_4$.

Entonces, siendo a_4 el ***ABA***, se invoca al método $\mathcal{M}_4$ de la *instancia* creada, con los parámetros $\mathcal{P}_3$, donde $a_4\ \vec{parametros}\ \mathcal{P}_3$. Como resultado de esta invocación se genera, en caso de que el método retorne algún valor, un resultado que es plausible de ser comparado con $\mathcal{R}_3$, donde $a_4\ \vec{pos}\ \mathcal{R}_3$, es decir el valor retornado almacenado en la **base de test de sistema**. En la figura 3 se presenta la situación desarrollada.

E.5 Caracterización de la secuencia capturada y la secuencia ejecutada

Al realizar una captura de un st_x, se realiza la captura de una secuencia de acciones. Estas acciones se capturan en orden secuencial y representan los estímulos que han recibido cierta *instancias* del *sistema* desde que es creada hasta que responde a cierto evento.

Figura 3. Ejemplo de Generación de Test de Unidad

Al crear un st_x, se mantiene esta serialidad, pero se realiza una **poda** importante de acciones. Dado que cada acción posee sus parámetros y su valor de retorno serializados, no es necesario almacenar los estados antes y después de cada ejecución de la acción debido a que las acciones son **autocontenidas**, es decir conservan los valores y son coherentes con los tipos de los parámetros de la invocación original. Esta estrategia, difiere de otras dado que **no conserva** los estados intermedios de las *instancias* [CDU2005], sino que los recrea **bajo necesidad**.

Nuestra propuesta pretende realizar un esfuerzo similar al que se realiza al crear un *Test de Unidad*.

Es por ello que luego de la poda realizada de acciones se logra justamente el esfuerzo artesanal que se realiza identificando solamente el ámbito de acción del método bajo análisis.

E.6 Contexto de Utilización

> *An effective way to test code is to exercise it at its natural boundaries. –Brian Kernighan*

Suponemos una baja dependencia de la *clase* $\mathcal{C}$ con otras *clases* para resolver la acción **ABA**. En particular, si las *clases* de las cuales depende han sido creadas por $\mathcal{C}$, estas *clases* serán recreados al generar el ut_x y podrán ser utilizados. Sin embargo, si las *clases* con las cuales interactúa la *clase* $\mathcal{C}$ han sido creados en el st_x por otras *clases* distintas de $\mathcal{C}$, estas *clases* no serán recreados al generar el ut_x dado que se ejecuta **sólo** la **secuencia restringida** a la *instancia* $\boldsymbol{c}$ de la *clase* $\mathcal{C}$, $\bar{\sigma}/\boldsymbol{c}$.

Ciertas *instancias* pueden emular sistemas de almacenamiento de información. Bajo *nuestra propuesta* si bien es posible acceder a estos sistemas de almacenamiento, el estado luego de *re-estimular* a la *instancia* puede no ser el mismo que se ha generado -por ejemplo, en forma colaborativa-. Este es el caso de Bases de Datos o *clases* ***singletton***, que pueden lograr cierto estado en forma colaborativa. *Nuestra propuesta* no ataca esta situación porque no es el ámbito de influencia de los *Test de Unidad*, dado que los *Test de Unidad* no pretenden verificar el comportamiento de la *clase* con otras *clases*. Para este tipo de verificaciones se utilizan *test de integración*, o los propios *Test de Sistema*.

Otras estrategias, como [SCR2006], presentan esta misma limitación mientras que [ATF2006] por ejemplo utiliza ***mock objects*** para emular a estas *instancias* tabulado ciertos valores, es decir, restringiendo las posibilidades de consulta a estos ***mock object***.

Nuestra propuesta supone una baja modificación de las interfaces[17] de las *clases*, es decir, de la signatura de los métodos. Esta restricción se debe a que al momento de *re-estimular* a la *instancia* para ejecutar el ***ABA*** los parámetros de las acciones de la secuencia $\bar{\sigma}$ deben coincidir en tipos y cantidad de parámetros, si existe diferencia en la signatura no será posible ejecutar esa acción. Este contexto, por lo general, sucede cuando una aplicación ha tenido una etapa de diseño de detalle rica o cuando la aplicación se encuentra en la etapa de mantenimiento de su ciclo de vida.

Asimismo, el contexto ideal de una *clase* $\mathcal{C}$, es aquella *clase* que no depende de otras *clases* previamente estimuladas. Un ejemplo de *clases* candidatas son aquellas *clases* que implementan *web services*, cuya invocación desencadena la creación de un contexto de ejecución y una respuesta a ese estímulo. *Clases* ***Stateless***, por ejemplo validadores de ciertas condiciones de negocio o *clases* de computos a partir de un *input* dado, son también representantes de *nuestra propuesta*.

Se supone que el generar los *inputs* de los *Test de Unidad* no es trivial. Esto puede ser por la coherencia que deben tener, o por el volumen de datos que reciben los métodos. En aplicaciones que utilizan modelos de datos dinámicos [WEBDBU] o invocación remota, los inputs de los métodos son objetos serializables, cuyo costo de generación es importante por su volumen y congruencia.

En un contexto de ***refactoring*** los resultados esperados de los métodos de las *clases* no varían. Pero sí el algoritmo que produce el resultado, por ello este contexto aprovecha la información en la **base de test de sistema** (que se supone estable) para verificar que el logro en la *performance*, por ejemplo, no ha afectado al cómputo que realiza el método.

F. Desafíos de esta propuesta

Una vez presentada la propuesta, presentaremos los desafíos inherentes para la implemetación de la misma para

[17] Una ***interfaz*** es una estructura de datos abstracta, la cual es utilizada para especificar la *interface* (en el sentido genérico del termino) que las *clases* deben implementar.

generar y verificar eficientemente los *Test de Unidad*:

- Dado cierto *sistema x* identificar las *clases* apropiadas sobre las cuales se puede y es interesante realizar *Test de Unidad* capturando la **secuencia de acciones** $\bar{\sigma}$ para cierto st_x.
- Utilizar la **secuencia de acciones** almacenada de cierto st_x para reproducir el ambiente de cierta *clase* $\mathcal{C}$ perteneciente al *sistema* $\mathcal{X}$ que ha sido modificado y generar el o los ut_x apropiados con injerencia en el cambio. De cada una de las **secuencia de acciones**, $\bar{\sigma}$, de ut_x, se debe extraer la información relevante a la simulación del componente, es decir la secuencia de acciones restringida al componente, $\bar{\sigma}/\boldsymbol{c}$.
- La captura del resultado y parámetros de cada acción no es una tarea sencilla dada la naturaleza del lenguaje. Por ejemplo, al utilizar *JAVA* muchos objetos no implementan la *interfaz java.io.Serializable* lo cual agrega un desafío adicional a ser resuelto.
- Para cierta acción capturada en el contexto de un st_x, y para cualquier ut_x tal que $st_x \longmapsto ut_x$, es nuestro desafío, poder realizar un análisis de diferencia del *resultado obtenido* al re-ejecutar en el ut_x la misma acción con respecto al correspondiente *resultado almacenado* en la secuencia de acciones del st_x.
- La modificación introducida en cierto $\mathcal{M}$ de cierta *clase* $\mathcal{C}$ puede conllevar a la imposibilidad de generar algún ut_x que involucre a $\mathcal{M}$. Identificar el contexto de ejecución de la *clase* $\mathcal{C}$ definirá la factibilidad de generar ut_x a partir de los st_x almacenados en la **base de test de sistema**.
- Por último, definir el ámbito de ejecución eficaz y alcance de esta estrategia es vital para aplicar el esfuerzo adecuado en el problema apropiado.

IV. Nuestra Estrategia

En el siguiente capítulo presentaremos *nuestra estrategia* **que implementa** *nuestra propuesta*(Capítulo III). Desarrollaremos la metodología, con la arquitectura, que resuelve los desafíos identificados en el sub-capítulo F del capítulo anterior. Analizaremos los aspectos de optimización de la propuesta presentado anteriormente, para generar los *Test de Unidad*. Identificaremos el alcance de *nuestra estrategia* y por último realizaremos una comparación con otras estrategias.

A. Estrategia de Adquisición de Datos

A continuación formalizaremos conceptos metodológicos a los efectos de presentar *nuestra estrategia* de adquisición de datos.

A.1 Evento

Un **evento** [OOS2004] es un suceso en el *sistema* (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El *sistema* maneja el *evento* enviando el mensaje adecuado al objeto pertinente.

A.2 Rango de Captura

Bajo nuestro modelo formal de captura de st_x, no se formaliza un comienzo y un fin de captura. De hecho, sería posible capturar una secuencia desde que cierto *programa* en ejecución lo permite, hasta que el *programa* finalice y no reciba más eventos.

A los efectos de implementar *nuestra estrategia* realizaremos la captura de un st_x desde que cierto *actor* del *sistema* genere un **evento**.

Este *evento*, por lo general, finalizará y será el mismo usuario quien verifique el *resultado obtenido*. A continuación definiremos el rango de captura mediante el concepto de **ejecución controlada exitosa**.

A.3 Ejecución controlada

Definiremos una **ejecución del sistema**, como la generación de cierto **evento** al *sistema*. Este evento es generado por un *actor* del *sistema*, que puede validar el comportamiento del *sistema* con respecto a lo que este usuario espera (especificación formal, manual de usuario, comportamiento deseado, entre otros).

La ejecución se realiza en un **contexto que aísla la operatoria** y permite diferenciarlo de otro evento. Esto es sumamente importante para luego poder analizar la ejecución del evento y capturar el st_x sin otros eventos de terceros.

Se podría permitir el acceso en simultáneo, pero dado que el problema que nuestra *tesis* ataca es la generación de *Test de Unidad* y éstos suponen una **baja colaboración con otras *clases***, *nuestra estrategia* simplifica esta interacción.

A.4 Ejecución controlada Exitosa

Diremos que una ejecución controlada es **exitosa**, si el *resultado obtenido* como resultado de la ejecución del **evento** se condice con el **resultado esperado** por el usuario. Es el usuario del sistema, quien al generar el evento y como respuesta a éste, valida el *resultado obtenido*.

A.5 Clases Observadas

Dado cierto *sistema* con un conjunto asociado de *clases* que lo componen, *nuestra estrategia* se concentra sobre un subconjunto de estas *clases*. En particular ciertas *clases* heredadas corresponden a *frameworks* [18] de terceros o *clases* del lenguaje de programación. *Nuestra estrategia* **sólo se concentra** en un subconjunto **seleccionado** de *clases* propias, llamaremos a este subconjunto de *clases* ***clases observadas*** o $\mathcal{CO}$.

La razón de restringir *nuestra estrategia* a un subconjunto de *clases* responde a una optimización del esfuerzo realizado y no a un limitante de la estrategia. Ciertas *clases* del *sistema* que no tienen lógica o cuya algoritmia no sea lo suficientemente interesante, no pertenecen al conjunto $\mathcal{CO}$ y no son observadas dado que no representan los problemas que esta *tesis* ataca. La selección de estas *clases* depende puramente del contexto y de los objetivos que se quieran alcanzar. En general en la etapa de diseño detallado ya se pueden identificar estas *clases* y sobre las cual se realizarán *Test de Unidad*.

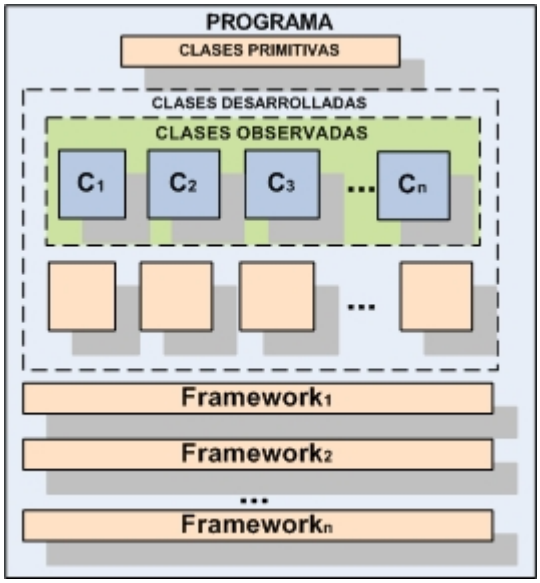

Figura 4. Clases observadas de cierto sistema.

A.6 Base de test de sistema

En el sub-capítulo D del anterior capítulo, hemos formalizado cómo *nuestra propuesta* construye una **base de test de sistema** a partir de la captura de la secuencia de acciones. Debido a que *nuestra estrategia* restringe la captura, entonces la **base de test de sistema** recolectará las *clases* **siempre restringidas** del conjunto de *clases observadas* $\mathcal{CO}$, $\bar{\sigma}/\mathcal{CO}$, durante la ejecución de cierta **ejecución controlada**.

A partir de esta **base de test de sistema** *nuestra estrategia* permite generar *Test de Unidad* para ciertas *clases* de las *clases observadas* $\mathcal{CO}$.

Es decir, que dada cierta ejecución del *sistema*, *nuestra estrategia* captura la secuencia de *acciones* **restringida a las *instancias* de las *clases observadas*** almacenando el correspondiente st_x. Para luego, cuando sea necesario, generar los ut_x que correspondan de los st_x almacenados.

B. Estrategia de generación de test de unidad

Dado cierta *clase* $\mathcal{C}$ de nuestro *sistema*, y cierto método $\mathcal{M}$, *nuestra estrategia* se vale de la información almacenada en la **base de test de sistema** para generar *Test de Unidad* ut_x para el método $\mathcal{M}$ de la *clase* $\mathcal{C}$.

En particular, si un desarrollador ha introducido un cambio en el método $\mathcal{M}$ obteniendo $\mathcal{M}'$. Entonces se pueden ejecutar los *Test de Unidad* para verificar que su cambio no afecta negativamente el comportamiento de la *clase* $\mathcal{C}$ en todas sus interacciones conocidas por la **base de test de sistema**.

Para verificar el cambio, la estrategia identifica el conjunto de st_x que en su secuencia contienen una acción que ejecute el método $\mathcal{M}$ de la *clase* $\mathcal{C}$.

Para cada uno de estos elementos del conjunto, se selecciona la *instancia* que ha ejecutado el método $\mathcal{M}$ de la *clase* $\mathcal{C}$, llamemos a esta *instancia* ***i***.

Se crea una *instancia* de la *clase* $\mathcal{C}$, y la estrategia implica crear una nueva *instancia* ***i'*** y *re-estimular* la misma, recreando la *instancia* original. La *re-estimución* se realiza con las acciones en el orden de la secuencia del st_x, restringidas a la instancia bajo análisis. Es decir, $(\bar{\sigma}/\boldsymbol{i})$, hasta llegar a la *acción* que invoca al método $\mathcal{M}$.

Luego de la invocación de este método $\mathcal{M}$ se verifica el *resultado almacenado* en la **base de test de sistema** con el *resultado obtenido*.

A continuación se brinda un *pseudo-algoritmo* que implementa la estrategia de generación de *Test de Unidad* para cierta *clase* $\mathcal{C}$ con cierto método $\mathcal{M}$ el cual ha sido modificado. (Ver Figura 5)

En la instrucción **3** de nuestro pseudo-código se toman todos los st_x que en su secuencia contiene al menos una *instancia* $\mathcal{C}$, que invoca al método $\mathcal{M}$, siendo $\mathcal{M}$ el método que ha sido modificado y cuyos *Test de Unidad* deseamos generar.

En la instrucción **4** se itera sobre este conjunto de st_x. En cada iteración se recrean las condiciones de cada st_x sobre la *instancia* ***inst*** creada en la instrucción **11** del pseudo-código.

Entre **13** y **22** se "*re-estimula*" el estado de la *instancia* ***inst***. Ante cada acción de la secuencia se verifica

[18] Un *framework* es una estructura de soporte definida en la cual otro proyecto de *Software* puede ser organizado y desarrollado. Típicamente, un *framework* puede incluir soporte a otros *sistemas*, bibliotecas y un lenguaje de *scripting* para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

```
1. C: Clase analizada
2. M: Metodo analizado
3. Cto:= Cjto. de test de sistema con injerencia en M
4. Mientras(Hay elementos(Cto)){
5.      //Son los mensajes de la sec. sobre C
6.      Secuencia s := SeleccionarUno(Cto)
7.      //Se obtiene la posicion en s de la ABA
8.      Entero pos := obtenerPosicionABA(M, s)
9.      //Se obtiene la accion ABA
10.     Accion aba := s[pos]
11.     Instancia inst := new Instance(C)
12.     Entero n := 0
13.     //Recreamos el ‘‘estado’’ de la instancia
14.     Mientras(n < pos){
15.        //Tomo la proxima accion de la secuencia
16.        Accion a := s[n]
17.        //Si la accion es de la instancia
18.        //bajo analisis se ejecuta la accion.
19.        Si( equivalenciaDeInstancias(a, aba))
20.            AplicarAccion(inst, a)
21.        n := n + 1
22.    }
23.    //Se ha ‘‘recreado’’ la instancia
24.    ReturnObject ro := AplicarAccion(inst, aba)
25.    Comparar(s[pos].resultado, ro)
26.    Quitar(s, Cto)
27.  }
```

Figura 5. Pseudo-Código: Algoritmo de generación de test de unidad para el método M de la Clase C

que la misma deba ser aplicada sobre la *instancia* que *re-estimulamos*. En particular el método *equivalenciaDeInstancias(a, aba)* en la instrucción **19** verifica que la *instancia* de la acción de *acción* de la secuencia ***a***, coincida con la *instancia* de la acción **ABA** bajo análisis. Si es el caso, entonces se aplica la acción a la *instancia*, ***inst***, creada anteriormente.

En la instrucción **24** se aplica la **ABA** sobre la *instancia* ***inst*** *re-estimulada*. De esta forma se ha finalizado la *re-estimulación* de la *instancia* y se ha ejecutado el método $\mathcal{M}$. Este método puede o no retornar algún resultado.

En la instrucción **25** se compara el *resultado obtenido* con el *resultado almacenado* en los datos de st_x. De esta comparación podría generarse algún informe para el desarrollador que ejecuta el *Test de Unidad*.

Nuestro pseudo-código recorrerá todos los st_x del conjunto ***Cto*** y luego finalizará su ejecución. A los efectos de simplificar el pseudo-código no hemos considerado el caso de múltiples **ABA** en el mismo st_x. Esto no reviste mayor complicación, dado que cada **ABA** representaría una ejecución completa del cuerpo del ciclo entre las instrucciones **4** y **26**.

B.1 Análisis de resultados obtenidos

Para el análisis de los resultados en la generación del ut_x, *nuestra estrategia* simplemente se reduce a verificar la igualdad de los objetos resultado. Esto se realiza mediante el método *equals* que debe tener implementado cada *clase* y verifica la *igualdad* de los objetos. Es responsabilidad del desarrollador o del lenguaje de programación (en nuestro caso *JAVA*) que este método este implementado en la *clase* y pueda ser ejecutado por la *instancia*.

En particular, y debido a que los objetos resultado pueden ser serializados se podría realizar una comparación a nivel *XML*[19], y presentar al desarrollador los resultados, como por ejemplo lo hacen [WEBXML] y una comparación de la serialización de objetos para *JAVA* [WEBDIF], que toma dos objetos y retorna un *array* con las diferencias.

Sin embargo el resultado más interesante que se obtiene al ejecutar una *suite* de *Test de Unidad*, es que éstos puedan ser ejecutados y no generen errores no esperados. Bajo este último caso, el desarrollador tiene la oportunidad de detectar su falla y repararlo. Ante un resultado fallido, el desarrollador podría realizar un *debug*, o *depuración*, de su cambio para analizar a qué se debe el fallo. Esta posibilidad es una fortaleza de *nuestra estrategia* dado que permite al desarrollador verificar la unidad con datos que bajo otro contexto pueden ser difíciles de generar.

Se podrían también generar heurísticas que permitan detectar tempranamente diferencias entre los *resultados almacenados* y los *resultados obtenidos*, cuando estos resultados comiencen a diferir, se podría dar por finalizado ese ut_x. Esta estrategia es más conservadora y reduciría drásticamente la aparición de falsos errores. Esta posibilidad se le puede brindar al desarrollador, quien puede seleccionar el tipo de test que ejecutará.

C. Alcance

> *I have not failed. I've just found 10,000 ways that won't work. –Thomas Edison*

La utilización de los test de **unidad generados** ut_x no tiene como finalidad el cubrir todas las funciones de los *Test de Unidad*. A continuación analizaremos el grueso de los casos que ataca esta estrategia e introduciremos la forma de extender esta estrategia.

La estrategia de captura de datos para la generación de *Test de Unidad* almacena la información relevante de estímulo de la *instancia* en ejecución para luego recrear su estado (inputs y output). La captura no pretende recrear todo el estado de la aplicación en ejecución, es por ello que no se captura, ni se recrea el estado anterior de las *instancias* con las cuales se podría interactúar.

Nuestra estrategia esta orientada a componentes *cross-cutting*[20] o servicios de infraestructura como ser *validadores*, reglas de negocio, *frameworks* de persistencia, de presentación y otros, que ante los mismos valores de entrada, sin importar la forma, deberían retornar el mismo resultado. Es decir, no dependen de un estado de otras *instancias* sino que la *instancia* bajo análisis lo crea de ser necesario.

Algunos ejemplos del alcance de *nuestra estrategia* son: la verificación de servicios de infraestructura (Ej: Persis-

[19] XML: Extensible Markup Language.

[20] En [CIA2005], se define *cross-cutting* cuando al realizar un *mapping* entre el *destino* y el *origen* de dos o más componentes, un elemento del *origen* esta relacionado con dos o más elementos *destino*.

tencia), reglas de negocio aplicadas en diversos procesos (business-level validation rules), *transformers*, *validadores*, entre otros.
En estos casos el contexto no depende de estados, sino de los valores de entrada de los métodos y lo que la *instancia* de la *clase* requiera para responder al pedido (Base de datos, otras nuevas *instancias*, *finders*, etc).

Por ejemplo una *regla de negocio* que verifica si un número de cuenta es válido o no, recibe valores de entrada específicos y sin importar los cálculos que realiza debería responder lo mismo a pesar de las modificaciones u optimizaciones realizadas por el desarrollador.

Esta *regla* es potencialmente invocada desde varias *clases*, dado que es una regla básica de negocio. Luego el método modificado u optimizado debería responder lo "mismo" que ha respondido en el contexto de una **ejecución controlada exitosa**.

En una aplicación comercial, este escenario es habitual, dado que se intenta aislar las validaciones, reglas de negocio, procesamientos compartidos. Se intenta entonces, que estos no dependan de un estado específico, sino que con ciertos valores de entrada reaccionen siempre generando el mismo resultado.

En general la **interfaz** de estos métodos es **estable** y se aplican sobre estos distintos "*refactories*" para mejorar su *performance* o reparar errores detectados. La estabilidad de estas *clases* es vital, dado que son la base de una aplicación, y los errores introducidos son difíciles de detectar y reparar. Por lo general los errores introducidos dependen de ciertos valores de entrada para "ocurrir" y ésto en general es difícilmente detectado al momento de desarrollar el código.

Volviendo a nuestro ejemplo de número de cuenta, si el método recibe un número de cuenta y debe responder un valor de verdad, se debe verificar que el número corresponda a un banco específico, que la sucursal de la cuenta exista, el tipo de cuenta, la moneda entre otros. Un cambio sobre esta verificación puede no tener en cuenta todas estas variantes e introducir un error que es difícil de detectar.

Otro aspecto importante en el desarrollo y mantenimiento es la mejora de *performance* en *Cross-cutting components* (DynamicBeans [WEBDBU] u otros elementos básicos).

Esto es crítico, dado que optimizar métodos es un proceso delicado y requiere de pruebas que generen un cubrimiento importante. No sólo para saber cómo se mejora la *performance* en las distintas variantes, sino para verificar el correcto funcionamiento, luego de la optimización.

C.1 Otras Estrategias y Comparación

Debido a la importancia de los *Test de Unidad* y su misión dentro del ciclo de vida del desarrollo del *Software*, distintas estrategias han surgido en los últimos tiempos.

En particular otras alternativas, como [CDU2005], utilizan los estados previos s_{pre} y posteriores s_{post} a la ejecución de la *clase* $\mathcal{C}$ y método $\mathcal{M}$ bajo análisis. Estas alternativa se denominan *state-based*. La estrategia entonces conserva el estado previo a la ejecución de **cada** acción para entonces poder realizar un "re-play" de la ejecución luego de haber incluído una modificación en el código.

En la figura 6 se puede apreciar cómo esta estrategia [CDU2005] realiza la captura de los estados en base a un *Test de Sistema* ejecutado por cierto usuario o sistema.

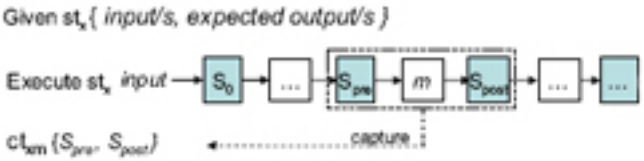

Figura 6. Captura de un test de sistema utilizando estados

Luego, en la figura 7 se realiza el *replay* o ejecución del *Test de Unidad* con la *clase* $\mathcal{M}$ modificada.

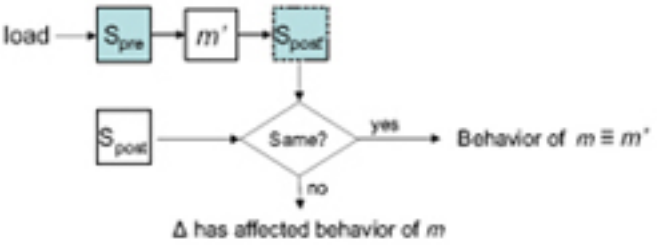

Figura 7. Ejecución de un test de unidad utilizando estados capturados

En general estas estrategias intentan reducir la captura de estados. Intentan identificar las diferencias entre estado y estado luego de ejecutar una acción, con el fin de optimizar el costo de almacenamiento de estados.

Por ejemplo [ATF2006], utiliza ***mock objects*** [WEBMAS] para emular a estas *instancias*, tabulando ciertos valores. Es decir, restringiendo las posibilidades de consulta a estos ***mock object***. Por ejemplo en la figura 8 se presenta un escenario donde se captura las invocaciones al modelo de datos. Esta estrategia no solo esta restringida al modelo de datos, sino que potencialmente podría capturar y simular otras *clases*.

Otra forma de generar *Test de Unidad* es mediante la estrategia [TIG2006], que realiza una ejecución simbólica de la *clase*, teniendo en cuenta los posibles valores que pueden tomar las *instancias* que son parte de la *clase* bajo análisis. De esta manera se pueden derivar valores de entrada para las unidades bajo análisis y reducir la redundancia de *Test de Unidad*. Esta estrategia [TIG2006] pretende optimizar los casos logrando cubrir los valores de entrada de estos *Test de Unidad*. Sin embargo, no se puede bajo esta estrategia, definir los resultados de los *Test de Unidad* que se generan y lo que se pretende es sólo realizar la ejecución, en lugar de validar la misma. De alguna manera esto es similar a la herramienta *JTest* [WEBJTE] que realiza ejecuciones simbólicas de *clases*, aunque [TIG2006] reduce los *test* redundantes.

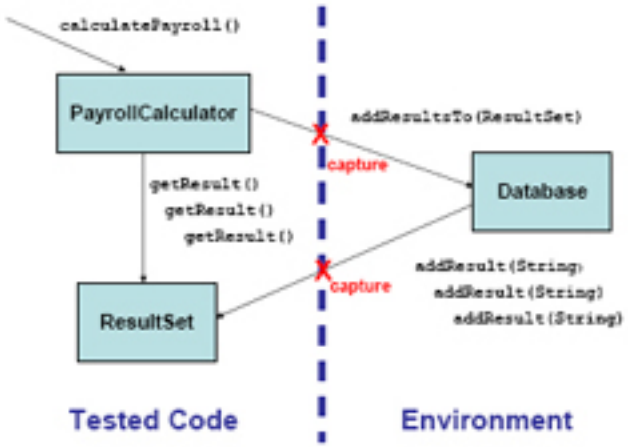

Figura 8. Captura de Interacción utilizando Mock Objects

Otras estrategias se basan en *stubs* [WEBMAS], es decir, una simulación del ambiente donde se ejecuta el objeto principal, brindandole -en forma acotada- la misma información que tenia en el contexto original de ejecución. Los *stubs* se pueden modelar como un diccionario que reemplaza la interacción en la simulación.

Por cada interacción de la simulación se crea una clave y una definición de la respuesta que brinda el "canal" que será simulado mediante el *stub*. Los *stubs* reemplazan a *instancias* que son onerosas y/o complejas en su creación o manipulación, como por ejemplo el modelo de datos.

La información en este diccionario es **finita** y **particular**. En el caso que el método modificado, $\mathcal{M}'$, varíe la "consulta", el *stub* no encontrará una definición para la nueva clave. El conservar todas las posibles definiciones representa una estrategia onerosa en volumen de datos y no es resistente a cambios en la interacción.

D. Justificación de la estrategia

Nuestra estrategia pretende que a igual secuencia de inputs el objeto se comporte generando un resultado "similar" a un resultado válido funcionalmente.

En otras palabras, si realizamos un *refactoring* de cierto método, logrando mejor *performance* ante cierto input, entonces deberíamos generar el mismo output. No obstante, no es cierto que necesariamente realicemos las mismas consultas, como lo suponen las estrategias basadas en *stubs*. Esta es una ventaja de nuestra estrategia, dado que no dependemos de un *stub* o diccionario acotado.

Asimismo *nuestra estrategia* es más ligera en cantidad de espacio dado que no conserva estados, sino invocaciones y el resultado de cada una de éstas para poder compararlo.

Este balance puede variar dependiendo del *sistema*, pero sin embargo el mero hecho de conservar cada estado al realizar cada acción indica una alta redundancia en lo que se conserva en el modelo de datos e indicaría preliminarmente que *nuestra estrategia* es menos costosa. Algunas estrategias como [CDU2005] intentar detectar estas diferencias entre acción y acción y minimizan los datos que se conservan, pero nuevamente no es una solución sencilla.

Si la *interfaz* es estable y no se modifica, luego no importa cómo el método $\mathcal{M}'$ (que difiere con $\mathcal{M}$) resuelva el problema, pero si importa que logre el "mismo" resultado. Esto es muy útil en sistemas que tienen afianzada su *interfaz* y que buscan optimizar sus cálculos.

El esfuerzo realizado para la captura de st_x en la **base de test de sistema** redunda en datos muy útiles para visualizar la utilización de las clases, como así también información relevante al desarrollador que modifica la *clase*.

V. Prueba de concepto de la estrategia

Dado que existen varios puntos que deben ser desarrollados para implementar *nuestra estrategia*, hemos optado por desarrollar una *suite* de herramientas específica para cumplir con cada objetivo de la implementación.

En este capítulo analizaremos cada uno de los componentes de la estrategia como así también los objetivos que implementa cada uno y la estrategia de la cual se vale para este fin.

A. Arquitectura Conceptual

En la figura 9 se presenta una arquitectura conceptual de *nuestra estrategia*.

Básicamente nuestra arquitectura se divide en dos grandes frentes. Un *subsistema* que captura los estímulos que recibe el sistema utilizando **AOP** [MAA2003] [21]. Luego serializa las *instancias* que son utilizadas como parámetros y resultado, conservando el orden de ejecución de las mismas en nuestra **base de test de sistema**. Esto es realizado utilizando *Hibernate* [WEBHIB] para permitir la persistencia de los datos.

El ***segundo subsistema***, se concentra en la generación de *Test de Unidad*. Como base es posible utilizar algún *framework* existente como *JUnit*, el cual es extendido con *nuestra estrategia*. Se utiliza nuevamente *XStream* [WEBXST] para la de-serialización de parámetros de los métodos que se invocan. Se utiliza un componente desarrollado por *nuestra estrategia* llamado *ClassLoad* que permite *re-estimular* cierta *instancia* a partir de un *Test de Sistema* almacenado.

En ambos subsistemas se utiliza el *sistema bajo análisis* para poder utilizar las *clases* desarrolladas y recrear las *instancias* necesarias o ejecutar los métodos a partir de las mismas. En los siguientes capítulos se profundizará más esta arquitectura llegando a la implementación de la misma.

B. Implementación de la Arquitectura Conceptual

Para realizar esta *prueba de concepto* de *nuestra estrategia* y generar una herramienta que verifique empíricamente la mejora al generar *Test de Unidad*, presentamos los siguientes *plugins* [PDP2003] de Eclipse:

- **JVirgilio**: permite mediante tecnología **AOP**, *Aspect Oriented Programming* [MAA2003] capturar en una secuencia los mensajes que intercambian las *instancias* de las *clases* en tiempo de ejecución.
- **JBeatriz**: permite acceder a los datos de los *Test de Sistema* que se han almacenado para cierta *clase* brindando visibilidad sobre los *test* almacenados.
- **JPythoness**: permite desde un test *JUnit* ejecutar diferentes *Test de Unidad* registrados para cierto método.

[21] AOP: Aspect-Oriented Programming.

Estos *plugins* están integrados a Eclipse, un IDE (Integrated development environment), [WEBECL] que soporta la construcción de *plugins* sobre esta plataforma para presentárselos al desarrollador.

La captura de información se realiza en el contexto de una ejecución controlada,st_x , donde la ejecución al finalizar debe ser considerada por el operador como "funcionalmente correcta". Si el operador decide que el st_x es funcionalmente correcto, será incorporada a la **base de test de sistema**. Potencialemente un st_x puede coleccionar la información necesaria para generar múltiples ut_x (Antes notado mediante el símbolo $\longmapsto$. Ver sub-capítulo E del capítulo III).

B.1 Run

Llamaremos a la ejecución funcional de un st_x realizada por un operador "***run***". El "***run***" posee un valor de verdad sobre el correcto comportamiento funcional de la ejecución del st_x asociado.

Un ***run*** genera ***casos*** para todas las *instancias* de objetos que ejecuta, y cada uno de estos ***casos*** posee ***datos*** que son los parámetros $\mathcal{P}$ de cada acción $\mathcal{M}$ y su respuesta.

B.2 Casos

Generalizando, podemos decir que el conjunto de *Test de Unidad* ut_x generados por *nuestra estrategia* para cada *instancia* $\boldsymbol{i}$, esta conformado por un conjunto de ***casos***, que son secuencias de mensajes que contienen como ABA seleccionada al método $\mathcal{M}$ del la *clase* $\mathcal{C}$, siendo $\boldsymbol{i}$ una *instancia* de la *clase* $\mathcal{C}$. Cada uno de estos ***casos*** ha sido ejecutado en el contexto de algún ***run*** funcionalmente válido.

B.3 JVirgilio

El primero de los componentes de la estrategia, es **Jvirgilio**, su nombre esta inspirado en Virgilio quien conduce al Dante en la **Divina Comedia** de Dante Alighieri. En esta obra Virgilio guía al Dante (alma de la Humanidad) por los círculos del infierno, es aquí donde Virgilio es guía y hace ver los detalles del infierno de las almas que han caído en desgracia.
La comparación no es casual, sino es parte de la metáfora de esta *suite* de herramientas. Las marañas internas del código son, en ejecución, difíciles de seguir, y es por ello que se requiere de un muy buen guía para poder seguirlas. Es el trabajo más arduo de la *suite* de herramientas, y de vital información para poder conformar esa **base de test de sistema** del sistema.

Jvirgilio es una herramienta desarrollada en lenguaje *JAVA* mediante Programación Orientada a Aspectos que permite capturar de manera secuencial los mensajes que intercambian los objetos *JAVA* en tiempo de ejecución. Esta herramienta propia, permite suscribirse a cierta *clase* $\mathcal{C}$ que se ha seleccionado y cuando ésta es "instanciada" se capturan sus estimulos y respuestas, obteniendo no solo los valores con los cuales es "instanciado", sino también los tipos de datos de cada interacción.

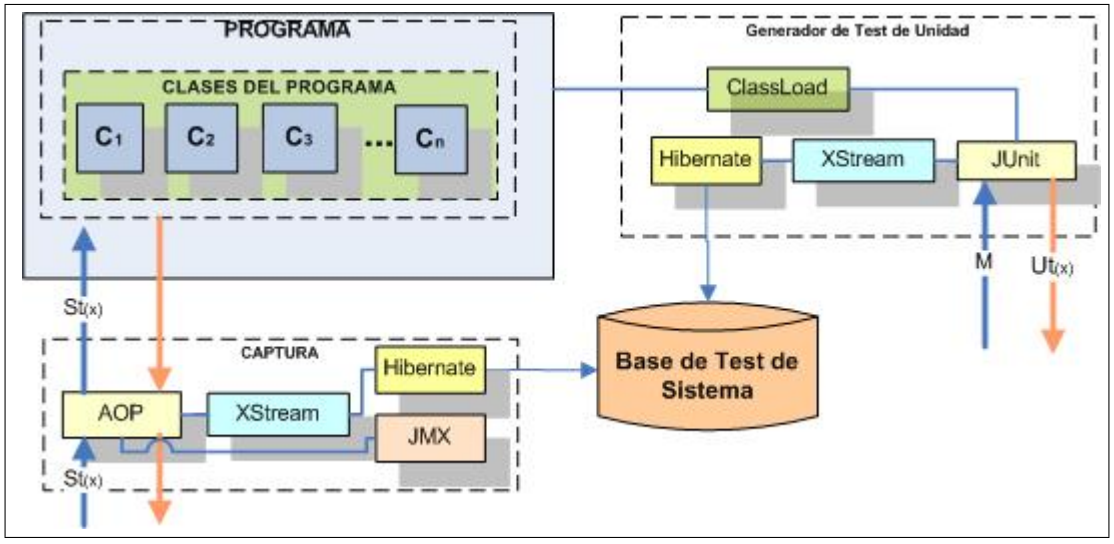

Figura 9. Arquitectura Conceptual de Alto Nivel de nuestra estrategia

La misión de **Jvirgilio** es guiarnos en las relaciones complejas del *Software* tomando estos mensajes para posteriormente ser analizados.

Arquitectura de la Herramienta

La arquitectura de **Jvirgilio** es simple. Esta herramienta se incorpora al Sistema que analizamos y expone una consola de Administración a quien utiliza la herramienta. Asimismo interactúa mediante un módulo de persistencia con una base de datos, almacenando la información relevante.

Utilización de la Herramienta

En primer lugar el sistema bajo análisis se le debe incorporar **Jvirgilio** a su "ejecutable" en forma de un **.jar** Una vez que este **.jar** se encuentre en el ***classpath*** de la aplicación a analizar, se inicia la aplicación y con está se iniciará la consola **JMX**[22] [WEBXMO] por medio de la cual podrá interactuar el operador de QA/QC[23].

Se ha utilizado una implementación estándar JMX, mediante HTML, cuyo *server* se activa en puerto *http* 8030. Este servidor es iniciado automáticamente cuando **Jvirgilio** detecta que se ha invocado aspectos relevantes.

Una vez que el servidor se ha iniciado, la primera acción del operador debe ser la de configurar el *sistema* que pretende utilizar mediante la consola **JMX**. En la figura 11 se puede observar el **method** que se expone al usuario **findOrCreateProgram** de esta manera el operador puede especificar qué *sistema* utilizará y **JVirgilio** tomará los datos asociándolos a este *sistema*.

Luego el operador está preparado para realizar operaciones sobre el ambiente y **JVirgilio** puede capturar los mensajes internos que utilizan las *instancias* de las *clases* para resolver la funcionalidad que ejecute el operador.

[22] JMX: Java Management eXtensions.
[23] QC: Quality Control.

Nuestro operador entonces realizará la ejecución de algún *caso de uso*, seleccionando ciertos datos y ejecutando el **evento** sobre el *sistema*.

Antes de realizar el **evento** sobre el *sistema*, el operador debe seleccionar este **evento** para que **JVirgilio** capture los mensajes de las *instancias* de las **clases observadas**.

A tal fin, debe accionar el **Start** de esta ejecución. Como se puede observar en la figura 12.

El operador debe introducir un texto representativo de la operación que va a realizar sobre el *sistema*. Esta descripción será presentada al desarrollador al momento de seleccionar los test representativos a su cambio.

Luego, el operador realiza el **evento** sobre el *sistema*, y es aquí, donde **JVirgilio** captura todos los mensajes relevantes internos que se envían las *instancias* de las **clases observadas**.

Luego de recibir la respuesta al **evento** que generó, el operador debe informar la finalización del **Run** para aislar la ejecución de otro **Run**. Para ello, el operador nuevamente utiliza la consola **JMX**, como puede observarse en la figura 13. Es aquí cuando el operador que domina la especificación de la respuesta esperada por el sistema, informa si el **Run** ha sido funcionalmente válido.

Luego, se ha capturado la información necesaria para generar casos de unidad sobre las **clases observadas** y están listos para ser utilizados.

En el lapso que ha transcurrido entre que el operador ha iniciado el **Run** y lo ha finalizado, el *sistema* potencialmente ha creado e invocado ciertas *instancias* de *clases* y es responsabilidad de **JVirgilio** el tomar las invocaciones sobre las **clases observadas** y conservarlas, serializándolas utilizando *XStream* [WEBXST] en el modelo de datos.

Estrategia de JVirgilio para la Persistencia de mensajes

Dentro de la ejecución del *sistema*, **JVirgilio** -mediante tecnología **AOP**- puede acceder a los mensajes que inter-

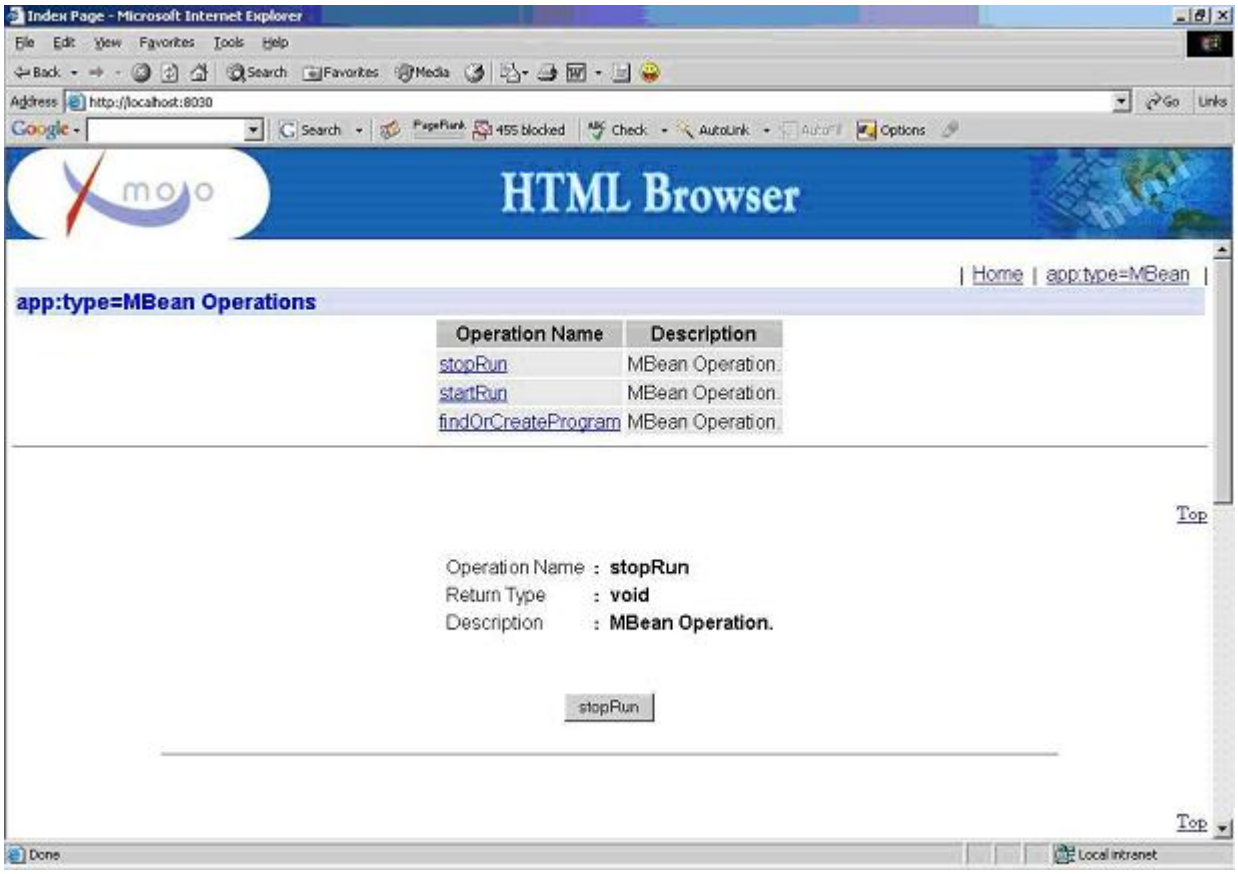

Figura 11. JVirgilio: Configuración del Sistema mediante consola

Figura 12. JVirgilio: Start del Run

cambian las *instancias* de las *clases observadas* colaborando en la solución al **evento**. Dentro de esta ejecución **JVirgilio** persiste esta información para luego ser utilizada por **JBeatriz** y **JPythoness**.

Un **Run** es iniciado por el operador mediante un **evento**. Este **Run** posee comienzo, fin y un resultado de éxito

Figura 13. JVirgulio: Stop del Run

de la ejecución. Dentro de este **Run**, **JVirgilio** crea los **casos de ejecución**. Es decir, todos aquellas invocaciones a métodos de *instancias* de **clases observadas** y sus respuestas que son necesarias para resolver la funcionalidad.

Estos casos son genéricos, en el sentido que sólo se enumera la ejecución y se formalizan los tipos que recibe cada uno de los métodos, pero no incluye los **datos** propiamente dichos.
Los **datos** o parámetros se persisten como tales, es decir como datos asociados a un *caso*.

Como se puede observar en 14 cada **Run** tiene asociado *casos* y *datos* de la ejecución.

Como se puede observar en el *diagrama de Secuencia* **JVirgilio**, figura 15, luego de iniciar el **Run** todas las invocaciones a *instancias* de *clases* del conjunto de **clases observadas** del *sistema* son captadas por **JVirgilio**. Este conjunto de **clases observadas** es *configurable* por el mismo operador o por el desarrollador. La granularidad puede configurarse en el mismo "**aspecto**" y no requiere otra acción ulterior dado que el modelo de datos soporta la persistencia de *instancias* serializadas.
La granularidad de las **clases observadas** también se puede dar por el nombre del *paquete*, nombre de la *clase* u otro.

Otros usos y Mejoras de JVirgilio

Debido a que **JVirgilio** representa un ejemplo de implementación de *nuestra estrategia*, el mismo puede ser mejorado y optimizado. A continuación hemos identificado algunos aspectos de mejora de la aplicación **JVirgilio**.

- La usabilidad de la consola **JMX** puede ser mejorada mediante un cliente ligero que sea más simple que navegar una página web.
- El desarrollador podría activar **JVirgilio** en su propio ambiente de desarrollo cuando es costoso regenerar el ambiente de la prueba. Luego, habiendo adquirido la información puede verificar su código con el *Test de Unidad* generado. Podría incluso compartir el *Test de Unidad* con otros desarrolladores o con una *base de test de sistema* global.
- La información que **JVirgilio** obtiene puede generar secuencias de ejecución "instanciadas" en ambientes donde es dificultoso generar estos diagramas. En particular, en ambientes donde la invocación es dinámica o donde esta configuración se realiza mediante archivos *XML* u otros, **JVirgilio** puede generar las secuencias en *run-time* o tiempo de ejecución.
- Con la información recolectada por **JVirgilio** en runtime, sería posible generar *diagramas de secuencia* para los *casos de uso* implementados. Es así, que el desarrollador, al finalizar el desarrollo de cierto *caso de uso* podría *re-estimular* al *sistema* mediante el *evento* que ejecuta su implementación y así recolectar la información apropiada. En el capítulo de Trabajos Futuros (Ver capítulo VI) presentamos esta funcionalidad.

B.4 JBeatriz

El segundo de los componentes de la estrategia es **JBeatriz**. **JBeatriz** lleva su nombre en homenaje a **Beatriz**, quien conduce al Dante por los Cielos en la Divina Comedia.

Beatriz, símbolo de la pureza, sirve de guía al Dante por los Cielos, donde a medida que uno camina más, los pasos son más fáciles de realizar y el conocimiento abunda. Esta misión ya no puede ser realizada por **Virgilio** quien sólo puede guiar por las penumbras del infierno y purgatorio.
Nuevamente la metáfora nos recrea en un ambiente donde

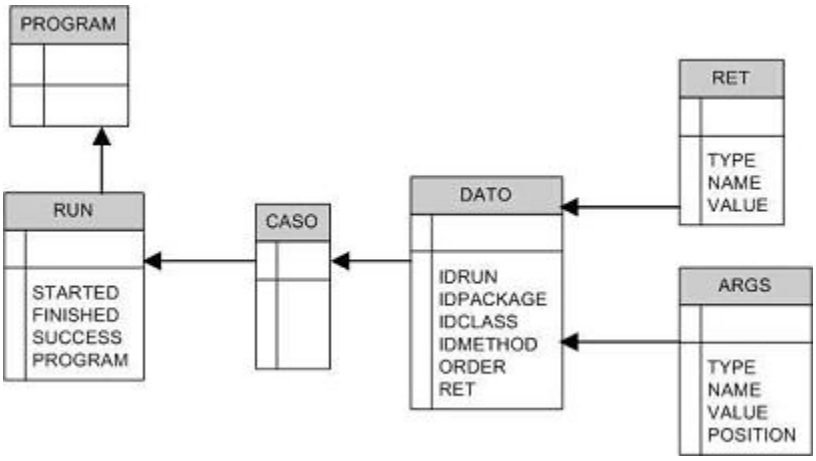

Figura 14. View simplificada del Modelo de Persistencia (RUN)

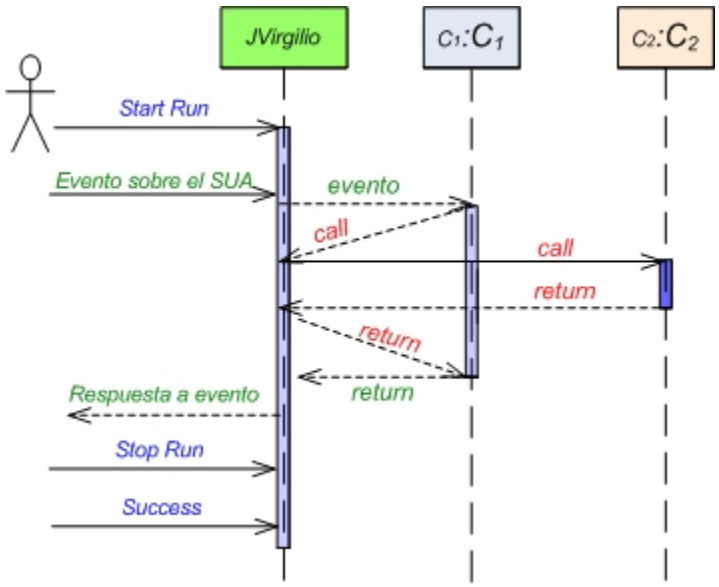

Figura 15. Diagrama de secuencia de ejecución y captura mediante JVirgilio

la información precisa **abunda**, y al desarrollador se le presenta información solo relevante a su búsqueda.

JBeatriz es quien interactúa con el desarrollador para que este "descubra" las *interfaces implícitas* que poseen las unidades. Asimismo **JBeatriz** le brinda datos de entrada representativos de la *clase* que el desarrollador consulta.

Arquitectura de la Herramienta La herramienta **JBeatriz** ha sido realizada bajo el **IDE**[24] **Eclipse**.

Esta herramienta esta integrada a la plataforma Eclipse en forma de *Plugin*.
En la figura 16 presentamos una Arquitectura conceptual del IDE Eclipse.

JBeatriz utiliza los servicios de soporte de Eclipse para presentar información relevante al desarrollador que realiza su proyecto en esta plataforma.
Es por ello que **JBeatriz** ha sido desarrollada utilizando tecnología **SWT** [WEBSWT] y **JFace** [WEBJFA] para presentar la información mediante una *interfaz* gráfica.
JBeatriz utiliza el *framework* de persistencia presentado en la Arquitectura de **JVirgilio** (Ver sub-capítulo B.3) para presentar información relevante al *sistema* que el desarrollador esta modificando.

Utilización de la Herramienta

La herramienta presenta un *panel* al desarrollador que contiene información sobre los objetos del *sistema* que desarrolla, como así también los *Test de Unidad* de cada *clase* que contiene la aplicación para cada *clase* en el conjunto de **clases observadas**.

JBeatriz presenta dos *views* de Eclipse, es decir dos componentes, la primera de ellas permite visualizar los *sistemas* de los cuales se posee registros, como así también las *clases* y *métodos* que contienen datos.

Como se puede observar en la figura 17 Detalle de la *view* de estructura **JBeatriz**, la información que se presenta del *sistema* es la misma estructura del *sistema*, es decir, su *paquete*, *clase*, *método* con los argumentos y tipo de retorno

[24] IDE: Integrated Development Environment.

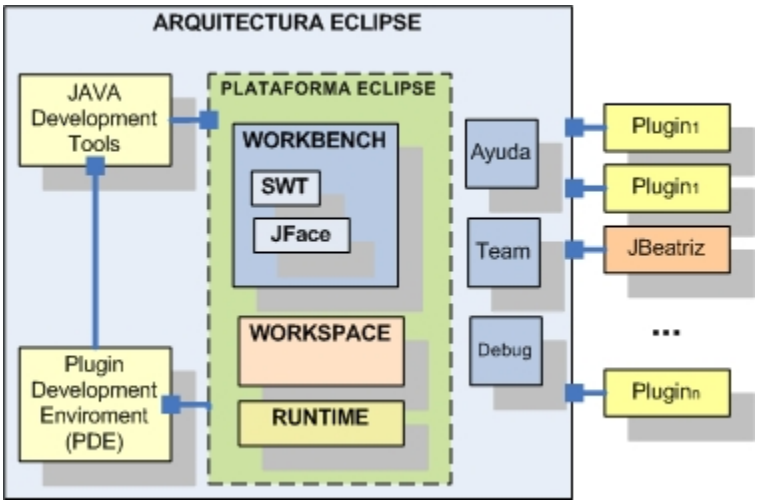

Figura 16. Arquitectura de Eclipse e integración con el plugin JBeatriz

de cada uno.

Es en este punto, donde se le brinda al desarrollador control sobre lo capturado por **JVirgilio** y puede eliminar información que considera innecesaria o que ha dejado de tener sentido dependiendo de su modificación en el código. Por ejemplo, si cierto desarrollador modifica la **signatura** de un método, entonces los *Test de Sistema* que capturaron la información de ese método no son útiles para este nuevo método (Ver Capítulo VI). Esto es debido a que no es posible *re-estimular* el método porque se han modificado(modificación de tipos de datos), agregado o quitado parámetros de la signatura. Y los *Test de Sistema*, serán "utilizables" **sólo** hasta el método que ha variado su signatura.

La función de esta **view** es presentarle al desarrollador el control total sobre la información que se encuentra en el modelo de datos. En definitiva es el desarrollador quien puede considerar la información existente como valiosa o prescindir de ella. Más aún, en un ambiente colaborativo, un grupo de test podría generar permanentemente información en el modelo de datos, pero serían en definitiva los desarrolladores quienes podrían seleccionar los *Test de Unidad* interesantes. Siempre compartiendo la **base de test de sistema**.

La segunda **view** que presenta **JBeatriz** al desarrollador es la **view de Run**. En esta view se presenta al desarrollador la información de cada **Run** almacenado por **JVirgilio**.
En la figura 19 se presenta la estructura de la view como así también los componentes que posee esta view.

Es aquí donde el desarrollador puede visualizar la secuencia de ejecución de cada uno de los **Runs** del *Sistema* que analiza, con la descripción funcional que ha ingresado el usuario de QA/QC.

El desarrollador también puede realizar aquí, la eliminación de los *Test de Unidad* (datos) recabados, por considerarlos obsoletos o poco representativos.
La *secuencia de ejecución* le brinda también información sobre los datos utilizados al invocar al método, es decir los valores que ha tomado la ejecución de ese Run.
Esta información es muy valiosa en los casos de aplicaciones dinámicas como se ha visto antes.

Configuración de JBeatriz

JBeatriz permite configurar (Ver figura 20) la interacción del Eclipse con la Base de Datos y otros parámetros de las herramientas. Esta información se persiste en el *core* de Eclipse, por lo que la configuración de la herramienta será personalizada en cada ambiente donde se ejecute **JBeatriz**.

B.5 JPythoness

JPythoness es la herramienta utilizada para verificar los cambios que se integran al código. **JPythoness** es invocado desde algún test *JUnit* u otra aplicación y ejecuta los test seleccionados por quien integra el cambio. **JPythoness** selecciona los casos y datos apropiados para la *clase* que es verificada y los presenta.

Luego, si es requerido, crea la *instancia* de la *clase* bajo análisis y *re-estimula* la nueva *instancia* en la forma apropiada para reproducir el estado de esta *instancia* y luego comparar los resultados que se han obtenido entre el **valor almacenado** y el **resultado obtenido**. Es en este punto, que se le presenta al desarrollador la diferencia -si esta existiese- y es el desarrollador en definitiva quien decide si el resultado del componente es apropiado y correcto.

Esta verificación, como se verá luego, no es trivial. Es el hecho de no poder predecir a ciencia cierta el resultado de la ejecución, quien da el nombre a la herramienta.

Ventajas de JPythoness

Los *Test de Unidad JUnit* son altamente conocidos en la industria de *Software*, sin embargo es complejo el generar *test de unidad* que representen el uso de la *clase* en cuestión y requiere de un amplio conocimiento. No sólo de la interacción con otros componentes, sino también las estructuras que la *clase* recibe como valores de entrada.

En un modelo donde los tipos de los valores de entrada

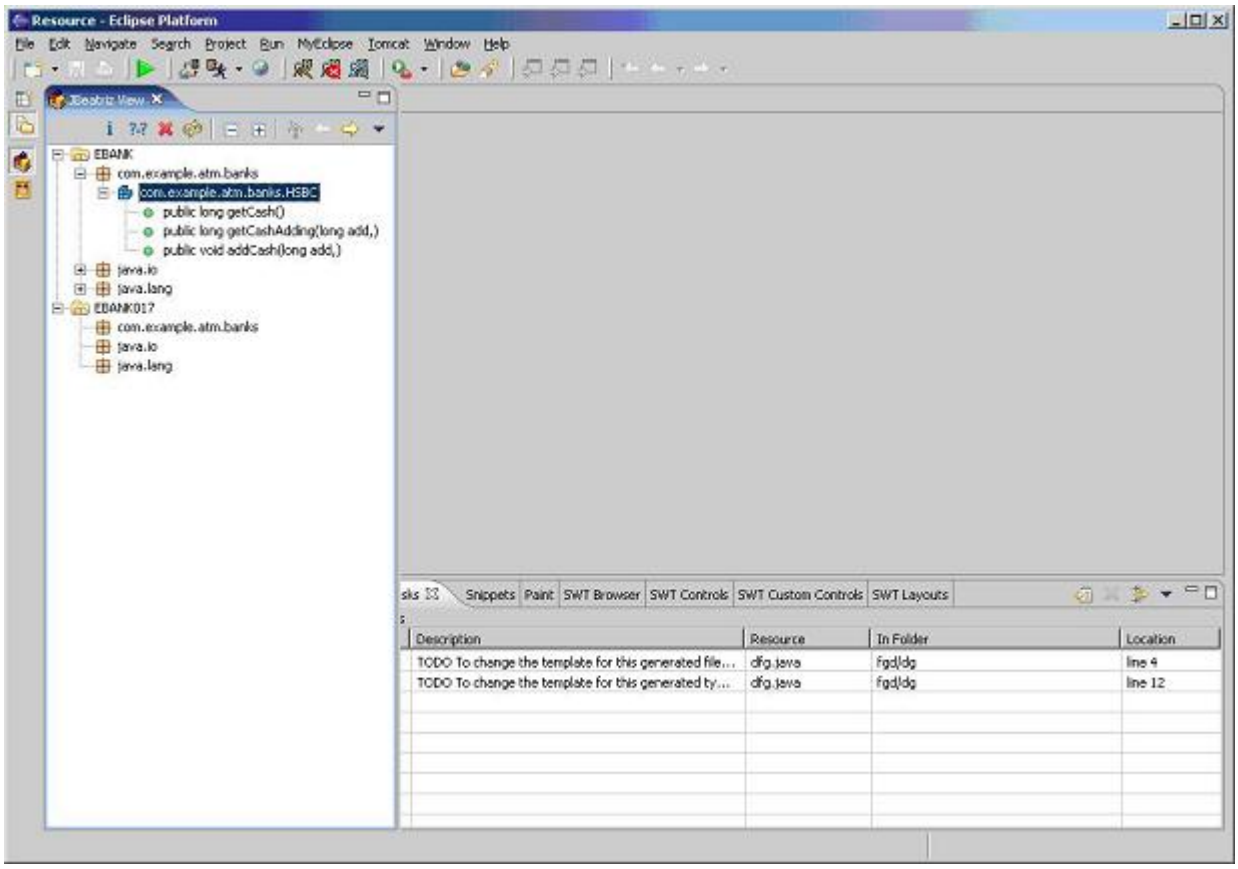

Figura 17. Detalle de view de estructura de JBeatriz

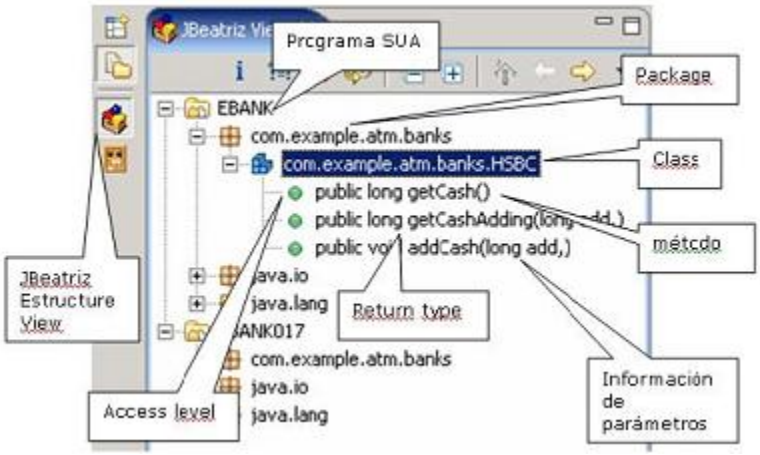

Figura 18. View de JBeatriz integrada a Eclipse

son conocidos y es sencillo generarlos, es sencillo realizar *test* que representen la utilización de la *clase*.

Por ejemplo si tenemos una función que recibe la dimensión de tres lados y responde un valor de verdad si es posible que las dimensiones formen un triangulo:

boolean esTriangulo (int a, int b, int c)

Luego podríamos establecer los casos y generar los datos en forma sencilla, aunque no siempre en forma completa [TAS2004].

Sin embargo cuando el método toma como parámetro un tipo de datos propio con una estructura compleja, el generar un dato de entrada puede ser una tarea no trivial. En particular en la aplicación para la cual hemos probado la herramienta los inputs de los métodos son *Dynamic-Beans* [WEBDBU] con estructuras representables en *XML* con varios niveles de anidamiento, complejas en si mismas y potencialmente el componente sólo afecta o es afectado por una parte de está estructura.

Por ejemplo, la operación que realiza la verificación de

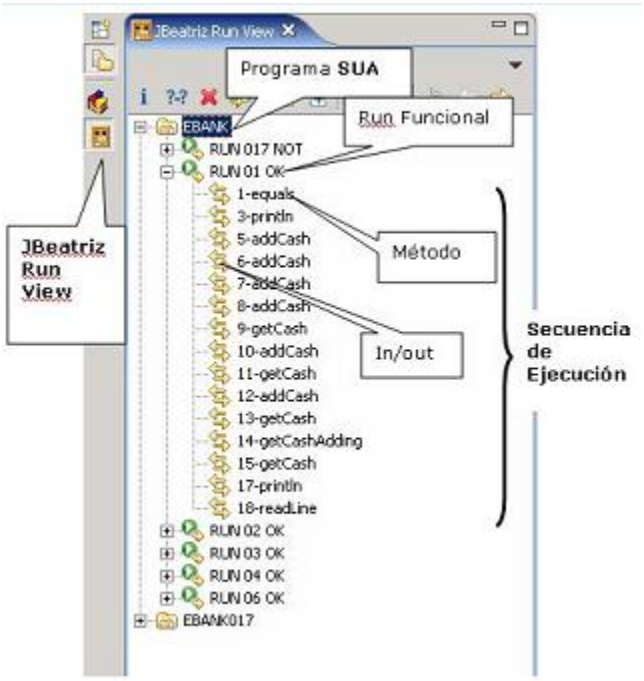

Figura 19. View de Run de JBeatriz

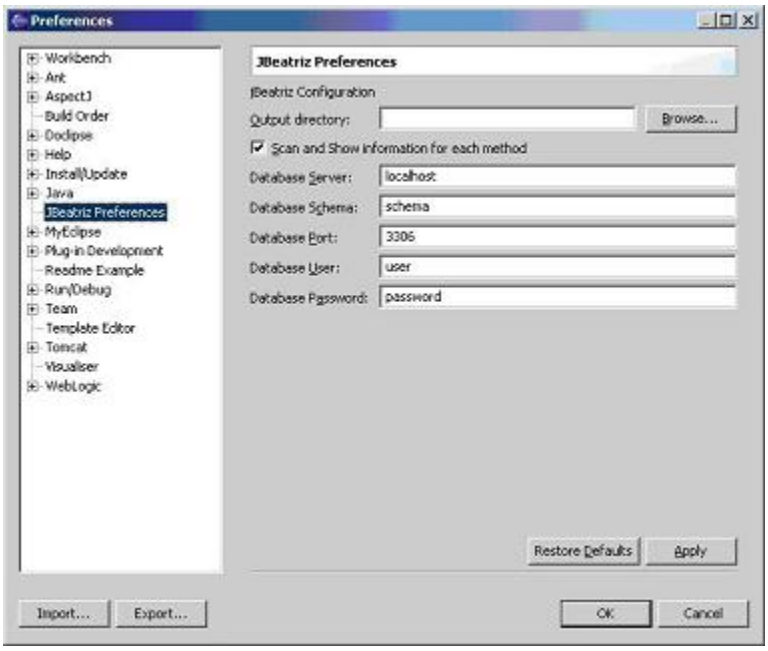

Figura 20. View de Configuracion de JBeatriz

la cantidad de veces que una persona ha desbloqueado su usuario recibe un *DynamicBean* del *Operador* con abundante información del mismo desde el modelo de persistencia, pero solo recupera una propiedad que es la cantidad de "desbloqueos" que ha realizado.
Si el *DynamicBean* no está bien formado, luego no podrá acceder al valor de esta variable y cancelará en tiempo de ejecución. Asimismo, si el método quita propiedades de este *DynamicBean* posiblemente afecte a un componente posterior que difícilmente sea detectado en tiempo de integración del cambio, y se deberá realizar un test de regresión importante para cubrir y detectar este error.

En una aplicación de *misión crítica*, un error en una *instancia* de QA/QC puede demorar la política de *releases*[25], más aún, el costo de detectar este mismo error y resolverlo sin introducir nuevos errores no es bajo.

Arquitectura de la Herramienta

JPythoness es una herramienta que "extiende" *JUnit*

[25] *Release* se refiere en este contexto a la liberación de cierta nueva versión de un paquete de *Software*.

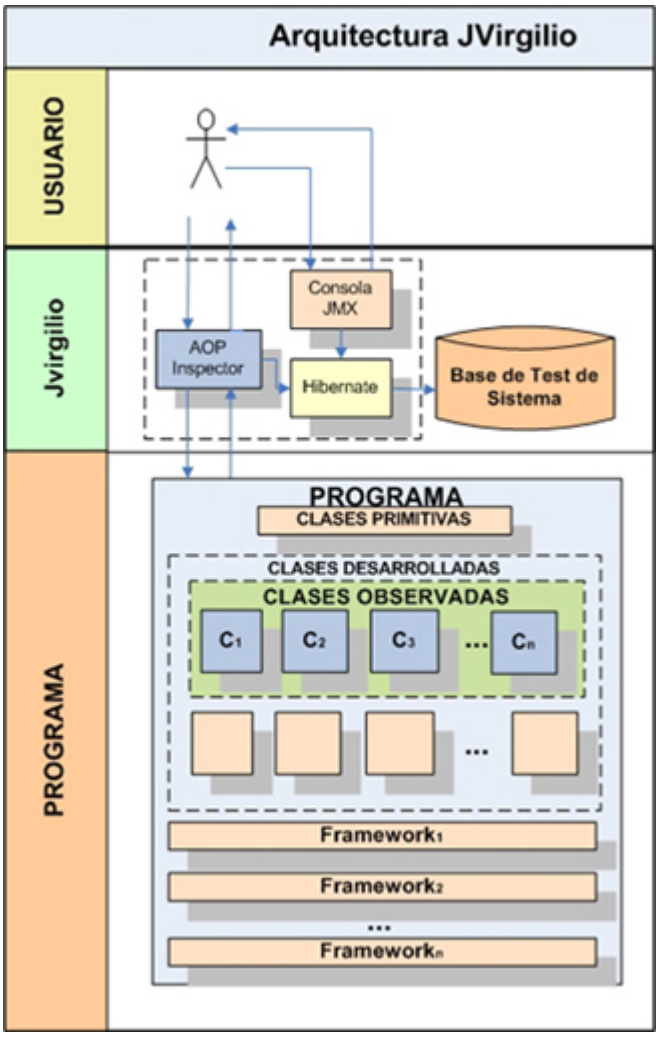

Figura 10. Arquitectura de alto nivel de JVirgilio

en su funcionalidad. La arquitectura de la herramienta es simple y contiene el mismo *framework* de persistencia que **JVirgilio** dado que utiliza el mismo modelo de datos.
Asimismo y dado que la intensión de la herramienta es interactuar con el desarrollador contiene un modulo de presentación desarrollado en *SWT* compatible con Eclipse.

Utilización de la Herramienta

La utilización de la herramienta está incorporada al test de unidad de *JUnit*, sin embargo es posible adaptarla a otros *frameworks* de *Test de Unidad* como ***cactus*** [JTF2002] u otros.

La invocación es sencilla, en el siguiente ejemplo

```
Pythoness pitonisa = new Pythoness();
pitonisa.testMethod(P,C,M,A);
```

Figura 21. Pseudo-Código: Ejemplo de invocación de JPythoness

Se realiza la verificación del método *M* de un cierto *programa* P. El test *JUnit* puede ser extendido con el método pitonisa.testMethod en alguna de sus sentencias y es así que invoca a la herramienta.

Es importante aclarar que esta funcionalidad se logra incorporando al proyecto el **pythoness.jar** que contiene a la herramienta.
La herramienta se vale del *path* y contexto de ejecución del test *JUnit* para realizar el load y creación de *instancias* de *clases* necesarias como así también "reproducir" el estado requerido para ejecutar cada *Test de Unidad* a partir del **Run** que se ha persistido en el modelo de datos.

Dado que para cada **run** en el modelo de datos, tenemos los mensajes que se le han enviado a este componente, la herramienta se encargará de regenerarlo en forma ordenada. Luego ejecutará el método sobre el objeto y verificará el *resultado obtenido* con el *resultado almacenado* en el modelo de datos.

La comparación no es trivial, y es por ello que es aquí donde el desarrollador tiene la vital tarea de definir si el cambio es correcto o no.

Si los resultados son los esperados, luego el cambio introducido no ha impactado o se ha compensado con otro, es decir, que a los efectos del *Test de Unidad* nada ha cambiado.
Sin embargo pueden suceder diferencias entre lo obtenido y lo esperado, para lo cual el desarrollador es quien mas conoce la *clase* que está modificando, por ende la persona competente para evaluar si la diferencia es significativa o se mantiene dentro de los valores esperados.

En todo momento el desarrollador es guiado por una *interfaz* que le permite seleccionar los *Test de Unidad* que ejecutará. La selección de los mismos se podrá realizar en base a la descripción funcional que ha ingresado el operador de QA/QC y se espera que estas reflejen el real comportamiento del *sistema*.

En la siguiente figura 22 se puede observar el resumen de **Runs** que se le presentan al desarrollador, es este, quien puede seleccionar los **runs** a utilizar para realizar el *Test de Unidad* del componente. Estos **Runs** son los que utilizan al componente en cuestión.

Estrategias de test de unidad sobre objeto

Como se ha visto en los anteriores capítulos, al momento de reproducir la *instancia* de la *clase*, para poder aplicar los mismos valores de entrada, existen varias alternativas:

- *Re-estimular* una nueva *instancia* de la *clase*
- Re-crear el estado anterior de la *instancia* y ejecutar el método bajo análisis
- Aplicar los valores de entrada sin tener en cuenta su histórico

Estas estrategias dependen de la situación en la cual la *instancia* de la *clase* se comporte en run-time.

Nuestra estrategia permite simular la primera situación mediante estimulación de los métodos necesarios sobre una *instancia nueva*.
En este caso se crea la *instancia* y se toman todos los mensajes que han involucrado a una *instancia* en ejecución y que ha capturado en orden **JVirgilio**, se aplican estas acciones y entonces la *instancia* está "disponible" para ejecutar el método que se verifica.

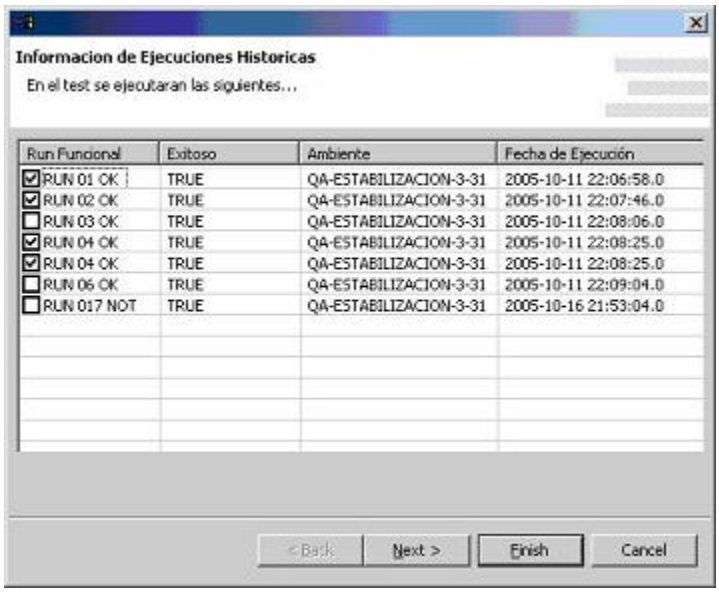

Figura 22. Selección de Runs

Nuevamente es el desarrollador quien puede configurar y definir la granularidad de la política de *re-estimulación* de la *instancia* para ejecutar el *Test de Unidad*.

Resultado de la ejecución del Test de Unidad

Una vez que se ha regenerado la *instancia* y se aplica el método en cuestión se obtienen resultados que han de ser "comparados" con el resultado que se había obtenido en la etapa de QA/QC y que ha sido capturado por **JVirgilio**.

La comparación en principio puede realizarse si el tipo de datos que devuelve el método es comparable (posee el método **equals**). Si es así, la comparación se realiza en los términos que el desarrollador lo ha estipulado al implementar el método **equals** de la *clase* o el **equals** en el caso de una *instancia* primitiva del lenguaje de programación (Como int, double, long en *JAVA*) como resultado del método.

Sin embargo muchas veces estos valores pueden diferir debido a que el modelo de datos ha "evolucionado" y los valores que antes se recabaron en la etapa de QA/QC ya se han modificado. En este último caso es útil conocer si el método respeta un contrato implícito.
Para tal fin, se puede comparar la taxonomía del objeto.

En particular la serialización de los parámetros y resultado es realizada mediante *XML* por **JVirgilio**, por ello aquí podemos valernos de esta taxonomía para realizar la comparación de los valores resultado. Si estos coinciden en estructura sabremos que nuestro cambio no afecta la taxonomia o meta-información que requiere el objeto y cuyo contrato es implícito.

Más estrategias de comparación pueden desarrollarse, más aún, podría predecirse los valores de referencia, pero ésto no se encuentra en el alcance de esta *tesis*.

B.6 Modelo de Datos de las Herramientas

Las herramientas utilizan el mismo modelo de datos dado que **JVirgilio** por ejemplo recolecta los datos que **JBeatriz** y **JPythoness** utilizan luego para evaluar un cambio, por ello se ha desarrollado un *framework* de persistencia con tecnología **Hibernate** que permite acceder eficientemente a los datos reaprovechando conexiones con un **pool de conexiones** [SEP2005] y modularizando toda la funcionalidad de persistencia y acceso al modelo de datos en un único componente reutilizable por las herramientas.

Se ha utilizado para este modelo la implementación de Base de Datos relacional de **MySQL** [WEBMYS] con soporte para transacciones.
En el Apéndice **MODELO DE DATOS** se encuentra una descripción de cada tabla con las entidades que modela cada una de estas.

En cuanto a la serialización de las *instancias* de los objetos en la base de datos se ha utilizado **XStream** [WEBXST] que consiste en una librería para serializar cualquier tipo de *instancias JAVA* a *XML* y vice-versa, ésto nos ha permitido conservar en el modelo de datos las *instancias* de las **clases observadas** y los parámetros que estas recibían.

C. Ejemplo simple de utilización

A los efectos de presentar prácticamente *nuestra estrategia* hemos desarrollado un muy sencillo ejemplo.

Presentamos entonces un modelo muy simplificado de un Banco y sus valores, con depósitos y algún método para conocer su estado actual de dinero. Nuestro *sistema* se llama **EBANK** y a continuación se presentan las características del mismo.

El modelo es muy sencillo y puede ser resumido en el diagrama de *clases* de la figura 23.

Como se puede apreciar nuestro ejemplo consta de un cajero automático (ATMMachine). Este cajero automático modela depósitos en un cierto Banco. En nuestro caso existe la *clase Bank* que modela justamente a este Banco y un Banco ejemplo con más servicios llamado *HSBC*

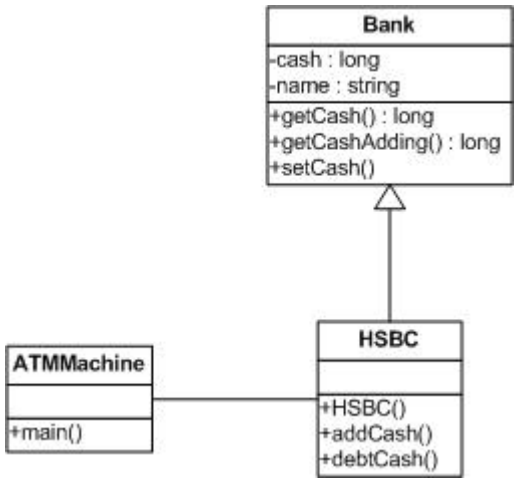

Figura 23. Diagrama de Clases - Ejemplo de Banco Simplificado

implementado en la *clase HSBC*.

En particular nuestro Banco *HSBC* permite agregar dinero al Banco mediante el método *addCash*, y debitar dinero del Banco mediante el método *debtCash*.

A continuación presentaremos el código fuente de la *clase HSBC* que implementa lo anterior:

```
package com.example.atm.banks;

public class HSBC extends Bank {
 public HSBC(){
        //Inicializo el Banco con 1000$
        this.setCash(1000);
        this.setName("HSBC");
        this.setManager("email@hsbc.com.ar");

    }

    public void addCash(long add){
        this.setCash(add + (this.getCash()));
    }

    public void debtCash(long debt){
        this.setCash(this.getCash() - debt);
    }
}
```

Figura 24. Clase HSBC

Luego el cajero automático modela justamente la interacción entre un posible cliente del cajero y el banco *HSBC*. Naturalmente esta relación, y a los efectos de presentar la idea, es sumamente simplificada.

Como se puede observar el método *main* del ATM crea un Banco *HSBC* y luego realiza depósitos sobre el Banco. Mientras realiza los depósitos invoca al método getCash() para conocer el estado de las arcas del Banco. Luego ejecuta el método getCashAdding agregando 10 unidades monetarias, y por último consulta el saldo de las arcas.

Al final se presenta al operador la posibilidad de volver a iniciar los depositos. En cada caso se crea un nuevo *HSBC*, ésto es a los efectos de poder activar y desactivar ***JVirgilio***, aunque podría ser un único Banco *HSBC* que

```
package com.example.atm;

public class ATMMachine {

 public static void main(String[] args) throws
 IOException {

  InputStreamReader i=new InputStreamReader(System.in);
  BufferedReader stdin = new BufferedReader(i);
  System.out.println("* BIENVENIDO ATM *\n");
  System.out.println("Command: ...");
  String line  = stdin.readLine();

  while(!line.equals("Q")){

     System.out.println("* BEGIN ATM SERVICE *\n");
     HSBC hsbc = (HSBC)new HSBC();

     hsbc.addCash(100);
     hsbc.addCash(100);
     hsbc.addCash(100);
     hsbc.addCash(100);
     long myCash = hsbc.getCash();
     hsbc.addCash(100);
     long myCash1 = hsbc.getCash();
     hsbc.addCash(100);
     long myCash2 = hsbc.getCash();
     long myCash3 = hsbc.getCashAdding(10);
     long myCash4 = hsbc.getCash();

     System.out.println("Y Cont. Q Salir?");
     line  = stdin.readLine();
   }
   System.out.println("***FIN***\n");

  }
}
```

Figura 25. Clase ATMMachine

evoluciona con el tiempo y el proceder de los clientes.

C.1 Utilizacion simple de JVirgilio y JBeatriz

Al ejemplo se le ha agregado la herramienta ***JVirgilio***. Se ha configurado el conjunto de **clases observadas** como todas las *clases* antes presentadas, luego se han realizado capturas de las ejecuciones.

Durante éstas capturas el sistema se comporta de la misma forma, pero ***JVirgilio*** observa detenidamente y persiste cada una de las acciones que se realizan sobre las **clases observadas**.

Luego de éste proceso, los datos capturados pueden ser utilizados y consultados mediante JBeatriz.

A continuación se presenta la vista de ***JBeatriz*** que presenta las **clases observadas** del *sistema* **EBANK**, con sus métodos, variables de entrada y tipos de objeto de retorno. Al presentar esta información el usuario podría consultar la información que se dispone de cada uno de estos métodos que han sido observados.

La próxima vista de JBeatriz (Ver Figura 27) presenta los *Run* que se encuentran en la Base de datos. Una descripción ingresada por quien ha capturado la secuencia de ejecución es útil a los efectos de conocer el caso de uso que representa u otro. En nuestro ejemplo hemos capturado 6

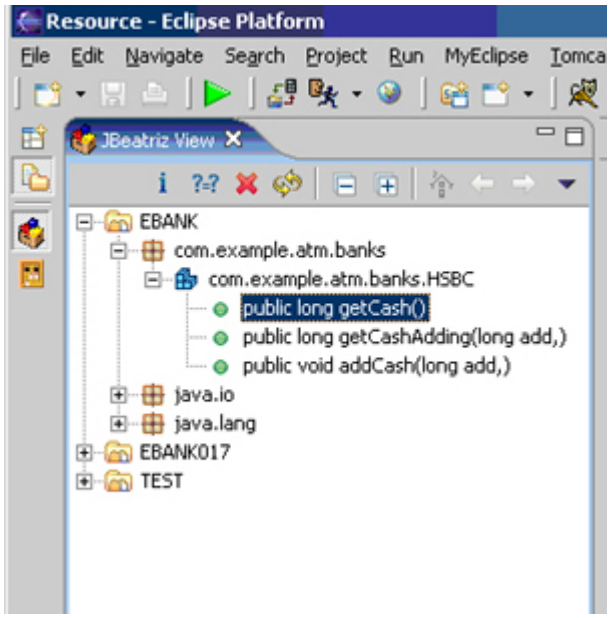

Figura 26. Consulta de Clases utilizando JBeatriz

Runs del *sistema* **EBANK**. Otros *sistemas* también disponen de ejecuciones, pero no expandimos esas ramas dado que no influyen sobre nuestro simplificado ejemplo.

Es en esta vista de ***JBeatriz***, donde el desarrollador puede decidir eliminar cierto *Run* (Ver Figura 28) de la **base de test de sistema**. O bien porque lo ha ejecutado y no ha sido útil, o bien porque el mismo ya no se encuentra actualizado.[26]

C.2 Utilización de JPythoness

Dado que modificaremos la *clase HSBC*, construimos un *Test de Unidad* que verifique la utilización de esta *clase*. Sin embargo no realizaremos código alguno para verificarlo, sino que será ***JPythoness*** quien verificará los *Run* que luego seleccionemos.

En la figura (Ver Figura 29) presentamos un test ejemplo de *JUnit* [JUN2001], el mismo ha sido extendido con *nuestra estrategia* y realiza las verificaciones sobre la *clase HSBC*.

En el contexto del test se crea una *instancia* de la *clase HSBC* y se la brinda a ***JPythoness*** para que ésta sea *re-estimulada*. [27].

La invocación se realiza como se presenta en el *pseudo-código* de la figura 30.

Donde **pro** es el *sistema bajo análisis*, en este caso **EBANK** [28]. La variable *clase* es la *instancia* que estamos verificando, esta información es utilizada por ***JPythoness*** para realizar el *classLoad* de la *clase*. La variable **method** representa el nombre del método que se desea verificar.

[26] En el capítulo VI de Futuros Trabajos presentamos una estrategia por la cual se verifican los cambios sobre la signatura de los métodos. Si la signatura ha sido modificada, se eliminan automáticamente todos los *Runs* afectados.

[27] JPythoness también podría crear la *instancia*, pero a los efectos de comprender este ejemplo lo hemos presentado de esta forma.

[28] Esto permite almacenar diferentes *sistemas* y relativizar las pruebas a cada uno de ellos.

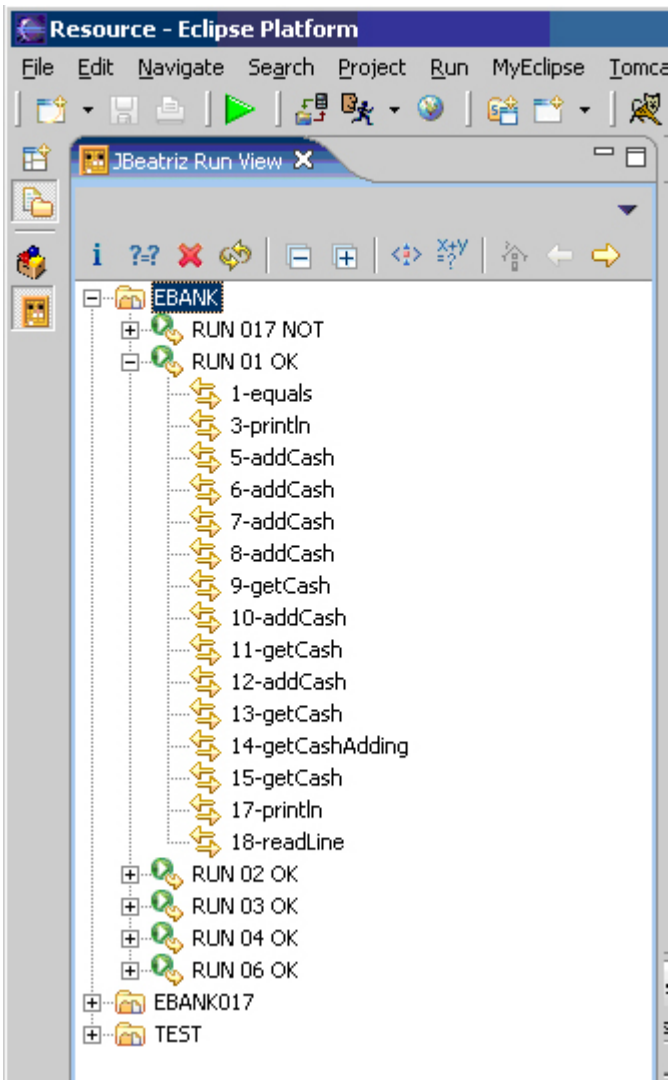

Figura 27. Ejemplo de Run visualizado mediante JBeatriz

Por último **vis**, la visibilidad del método, en nuestro caso este es *public*.

De esta forma los *Test de Unidad* existentes se pueden integrar a los que realiza ***JPythoness***.

El método bajo análisis en esta caso es ***getCashAdding***. Este método es heredado de la *clase Bank*, recibe un *long* como parámetro, lo suma a las arcas y luego retorna la cantidad de unidades monetarias de las arcas del Banco.

En la figura 31 se presenta un ejemplo de cómo invocar al *Test de Unidad* antes presentado.

Luego de invocar el *Test de Unidad*, ***JPythoness*** presenta la siguiente pantalla (Ver Figura 32)para seleccionar los *Test de Unidad* que son relevantes:

Luego ***JPythoness*** ejecuta los *Test de Unidad* seleccionados. En cada caso *re-estimula* una *instancia* nueva de la *clase HSBC* y presenta por consola acción a acción las invocaciones como así también los resultados de los métodos.

A continuación (Ver Figura 33) se presenta parte del código fuente que se ejecuta para *re-estimular* la *clase*

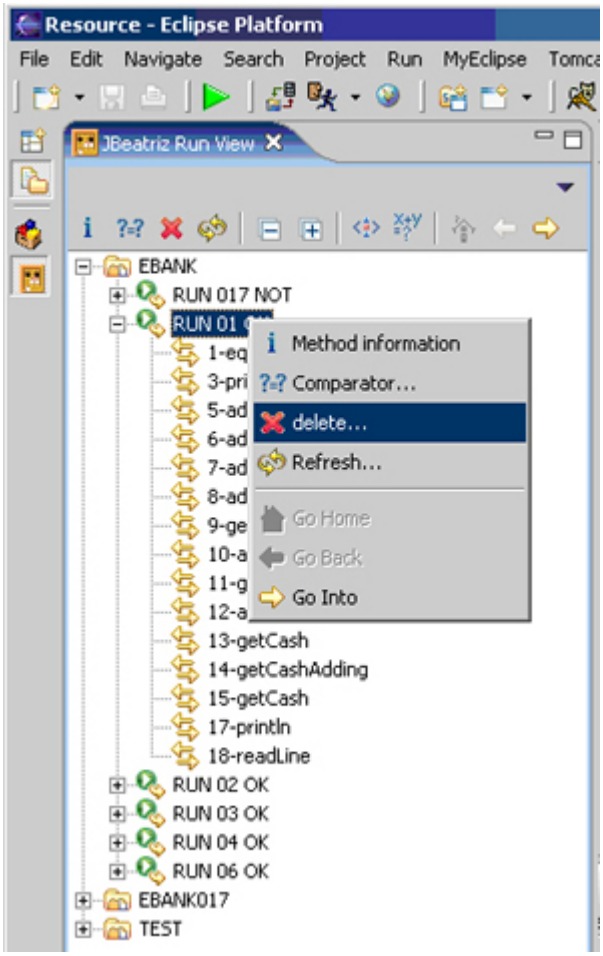

Figura 28. Eliminación de Run mediante JBeatriz

```
package com.example.atm.banks;

import junit.framework.Assert;
import junit.framework.TestCase;
import ar.uba.dc.pythoness.Pythoness;

public class HSBCTest extends TestCase {

 private HSBC clase = new HSBC();
 private String pro = "EBANK";

 public static void main(String[] args) {
  junit.swingui.TestRunner.run(HSBCTest.class);
 }

 protected void setUp() throws Exception {
  super.setUp();
 }

 protected void tearDown() throws Exception {
  super.tearDown();
 }

 public void testGetCashAdding() {

  Pythoness pitonisa = new Pythoness();
  String method = this.getName().replaceAll("test","");
  String vis = "public";
  pitonisa.testMethod(pro,clase.getClass(),method,vis);

 }

}
```

Figura 29. Clase HSBCTest que implementa un JUnit extendido con JPythoness

HSBC:

Una vez que se ha *re-estimulado* a la *instancia* de la *clase HSBC*, se invoca al método bajo análisis, ***getCashAdding***, y por último se realiza la comparación del *resultado obtenido* con el *resultado almacenado* (Ver Figura 34).

Por ejemplo, luego de ejecutar el test sobre *HSBC* se obtiene el siguiente resultado en la consola de la aplicación (Ver Figura 35):

Como se puede apreciar al final de la captura de la consola presentada en la Figura 35, se ha *re-estimulado* a la *instancia HSBC* con los métodos relevantes del *Run* a la *instancia* (incluyendo los métodos que ésta heredaba de la *clase Bank*) teniendo en cuenta el método heredado ***getCashAdding***.

Al reejecutar el *Run* se logra el mismo *resultado obtenido* que el *resultado almacenado* oportunamente, el *long* **1610**. Recordemos que la *clase* HSBC es inicializada con sus arcas en 1000 unidades monetarias, por ello y luego de 6 depósitos de 100 unidades más las 10 últimas unidades el total es **1610**.

Dado que el *long* es un valor primitivo, es posible realizar la comparación y ***JPythoness*** informa este resultado como se puede apreciar en la figura 35.

```
Pythoness pitonisa = new Pythoness();
String method = this.getName().replaceAll("test","");
String vis = "public";
pitonisa.testMethod(pro,clase.getClass(),method,vis);
```

Figura 30. Invocación de JPythoness

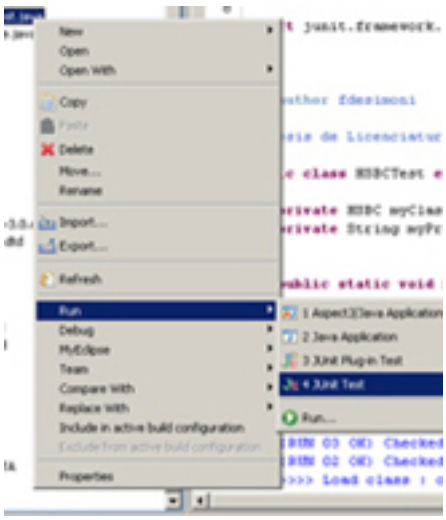

Figura 31. Inicio de Ejecución de Run utilizando JPythoness

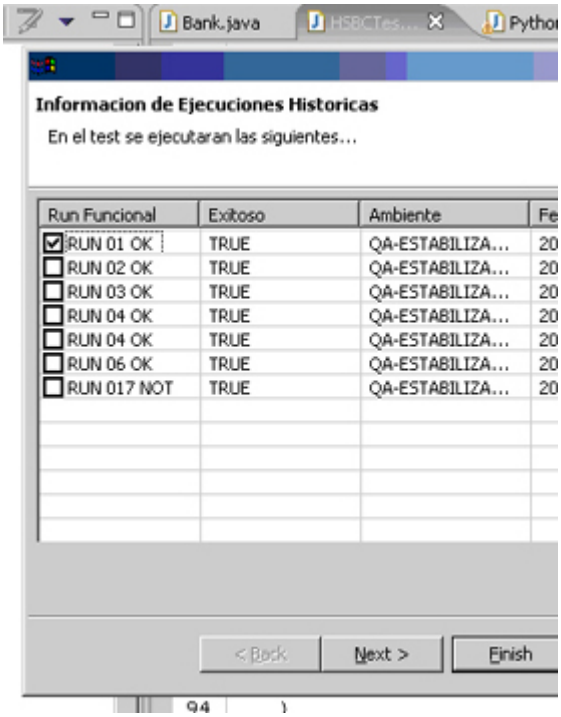

Figura 32. Ejecución de los Run seleccionados utilizando JPythoness

```
...

public static Object loadClassWithRegresion
(Session ses, HData theData){

 //Creo el Objeto que necesito dependiendo de data
 HMethod theMethod = (HMethod) theData.getMyMethod();
 String className=
     theMethod.getMyClass().getDescription();

 //Cargo el objeto
  Object obj = ObjectFactory.loadObject(className);

 //Busco los methods que aplicaron(Write) al class
 List orderedMethods=
     findPreviousDataToData(ses,theData);

 //Por cada "data" de la List debo ejecutar
 //ese method sobre la instancia del Objeto
 int size = orderedMethods.size();
 for(int i = 0; i< size; i++){
   HData data = (HData) orderedMethods.get(i);
   //El ret de la execucion del method
   //no importa en este caso.
   Object ret=ObjectFactory.executeMethod(obj, data);
   System.out.print(".");
 }
 return obj;
}

...
```

Figura 33. Re-estimulación de la instancia

```
result=compareResults(myStoredResult,myRegresionResult);
```

Figura 34. Comparación de Resultado Obtenido con Resultado Almacenado de getCashAdding

```
   >>>>>> Load class : com.example.atm.banks.HSBC
   >>>>>> returning system class (in CLASSPATH).
EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL m\'{e}todo: 100
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL m\'{e}todo: 100
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL m\'{e}todo: 100
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL m\'{e}todo: 100
.EJECUTA metodo:
public long com.example.atm.banks.Bank.getCash()
ARGUMENTOS DEL m\'{e}todo:
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL m\'{e}todo: 100
.EJECUTA metodo:
public long com.example.atm.banks.Bank.getCash()
ARGUMENTOS DEL m\'{e}todo:
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL m\'{e}todo: 100
.EJECUTA metodo:
public long com.example.atm.banks.Bank.getCash()
ARGUMENTOS DEL metodo:
.        >>>>>> Load class : long
        >>>>>> returning primitive class.
EJECUTA metodo:
public
long com.example.atm.banks.Bank.getCashAdding(long)
ARGUMENTOS DEL metodo: 10

RESULTADO EQUIVALENTE:
Almacenado->1610 Obtenido->1610
```

Figura 35. Comparación de Resultado Obtenido con Resultado Almacenado

C.3 Modificación de HSBC y verificación del cambio

En el anterior ejercicio, se ha presentado cúal es la mecánica para ejecutar un *Test de Unidad* bajo *nuestra estrategia*.

A continuación, se introducirá un cambio sobre un método de la *clase Bank* (método ***getCashAdding***) y se ejecutará el mismo *Test de Unidad*. Como el método es heredado por *HSBC*, entonces el mismo *Run* antes ejecutado es relevante a la verificación del cambio introducido.

Entonces, supongamos que se realiza el cambio, lográndose en la *clase* **EBANK** como se presenta en la figura 36.

```
public long getCashAdding(long add) {
   this.cash = (this.cash + add)/2;
   return this.cash;
}
```

Figura 36. Modificación introducida en el método getCashAdding de la clase Bank

Nótese que el cambio introducido , que se presenta en la figura 36, modifica el valor que se conserva en las arcas del banco en cuestión. Es en este punto, donde al invocar a este nuevo método se incorpora el depósito, pero el valor de las arcas con este depósito es dividido por dos.

Claramente ésto no responde a lo que uno desearía realice un método que agrega dinero a las arcas, entonces veamos como reacciona el mismo *Test de Unidad* antes ejecutado mediante ***JPythoness*** con este nuevo cambio.

Para verificar el cambio no es necesario realizar acción alguna, simplemente es necesario compilar la nueva *clase* [29] y re-ejecutar el test HSBCTest.

Luego de ejecutar el *Run* ***RUN 01 OK*** el resultado de la consola se presenta en la Figura 37.

```
    >>>>>> Load class : com.example.atm.banks.HSBC
    >>>>>> returning system class (in CLASSPATH).
EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL metodo: 100
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL metodo: 100
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL metodo: 100
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL metodo: 100
.EJECUTA metodo:
public long com.example.atm.banks.Bank.getCash()
ARGUMENTOS DEL metodo:
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL metodo: 100
.EJECUTA metodo:
public long com.example.atm.banks.Bank.getCash()
ARGUMENTOS DEL metodo:
.EJECUTA metodo:
public void com.example.atm.banks.HSBC.addCash(long)
ARGUMENTOS DEL metodo: 100
.EJECUTA metodo:
public long com.example.atm.banks.Bank.getCash()
ARGUMENTOS DEL metodo:
.        >>>>>> Load class : long
        >>>>>> returning primitive class.
EJECUTA metodo:
public
long com.example.atm.banks.Bank.getCashAdding(long)
ARGUMENTOS DEL metodo: 10
DIFIERE RESULTADO:
Almacenado->1610 Obtenido->805.
```

Figura 37. Resultado de la modificación introducida en el método getCashAdding de la clase Bank

Nótese que ***JPythoness*** presenta en la figura 37 la diferencia, informando que el *resultado obtenido* es **805**, mientras que el *resultado almacenado* para esta ejecución es **1610**. Justamente **805** es la mitad de **1610**.

[29] Eclipse permite hacer esta compilación automáticamente, por lo que sólo se debe modificar la *clase* y luego ejecutar el test automáticamente.

C.4 Conclusiones del ejemplo simplificado

Sin duda el ejemplo es muy simple, y no pretende mostrar la robustez de la estrategia, sino un ejemplo práctico de su uso y como -por ejemplo- soporta la herencia de métodos entre *clases.*

La modificación introducida es muy simple de detectar y no presenta mayores problemas, sin embargo no siempre es el caso, más aún en validaciones complejas que podría realizar el getCashAdding, como por ejemplo el número de cuenta, titularidad de la cuenta u otra validación de base. Este simple ejemplo presenta la factibilidad de la estrategia y los resultados que se obtienen rápidamente con *Run* almacenados relevantes al cambio.

Mucha de la interacción al ejecutar el *Test de Unidad* podría automatizarse para que el desarrollador no deba seleccionar los *Run.* En el capítulo de Futuros Trabajos se presentan incluso las debilidades de la *interfaz*, dado que esta debe resumir eficientemente los resultados de la ejecución de los *Test de Unidad.*

D. Caso de Uso Práctico y Proceso de Utilización

Presentaremos cóomo *nuestra estrategia* es incorporada al proceso de una aplicación real. Optimizando el control sobre los *RFC* [30] para un contexto definido. Este caso real ha inspirado nuestra estrategia y consideramos que es un buen reflejo de la realidad que *nuestra estrategia* desea cubrir.

Durante la concepción de esta tesis se ha investigado la problemática de una aplicación financiera -actualmente en producción- que realiza operaciones por más de **300 millones de dolares mensuales** con **1.500.000 de transferencias mensuales** y con más de 45.000 empresas clientes.

El foco está en dar control al desarrollador que incorpora un cambio a esta aplicación dominando el impacto del cambio que introduce y el alcance. Verifica en cada caso, que la solución propuesta no impacta en forma no esperada en otra área de la aplicación.

Las aplicaciones que verán optimizado su control frente al cambio son aquellas que **reutilizan componentes**, en particular creemos que los componentes altamente utilizados como servicios de infraestructura , *rules* y cálculos de verificación son los más beneficiados y es nuestro foco de atención.

Contexto

La aplicación financiera que inspira la estrategia está desarrollada en el lenguaje orientado a Objetos *JAVA*, pero intervienen en la arquitectura otras tecnologías, a saber:

J2EE [WEBJAV] como container para administrar servicios que requieren un nivel de robustez y seguridad elevado como así performance especificada.

Xdoclet como generador de código automático para interfases.

Struts como modelo MVC para presentar funcionalidad

[30] RFC: Request for Change.

mediante una *interfaz* Web
XWorkFlow para generar dinámicamente distintos workflows de negocio.
Framework propio de generación de consultas y persistencia, como así también un *framework* de presentación específico para el dominio de aplicación.

Asimismo se dispone un *framework* de test que permite utilizar los servicios de la aplicación pero el uso de este *framework* no es generalizado y **sólo algunas *class* tienen test asociados**.

Se utiliza un mecanismo de comunicación entre objetos mediante **DynamicBeans** [WEBDBU], estos objetos constituyen el pilar de la comunicación explicitando en cada caso meta-información del contenido del mensaje. Es decir información sobre los tipos, su nombre y su valor, como así también estructuras más complejas anidadas. La **comunicación entre los diferentes objetos es implícita**, es decir, **no existe un contrato explícito** entre las *instancias* que interactúan, y es responsabilidad del desarrollador construir *clases* que interactúen armoniosamente.

La aplicación contiene su modelo de dominio en una base de datos relacional. Existe a su vez una dependencia fuerte entre cada *release* y su **base de datos**, dado que las tablas de esta base de datos dan soporte al *framework* de persistencia. En este contexto existen distintas bases de datos de diferentes ambientes: *Producción, QA, Testing, Desarrollo y Performance*. A estas bases de datos se le aplican scripts continuamente migrándolas para soportar la nueva *release*.

El contexto de desarrollo de la aplicación de *misión crítica* es bajo el ciclo de vida ***Waterfall***. Con una fuerte etapa de concepción, verificación de los requerimientos, arquitectura, diseño, implementación y verificación de lo implementado.
El ciclo de vida se desarrolla como se presenta en la Figura 38 Ciclo de desarrollo de *Software*.

Dentro de este contexto de desarrollo de *Software*, encontramos diversos equipos con funciones definidas.

D.1 Concepción de Requerimientos

En primer lugar un equipo denominado **Gerencia de Producto**, coordina con los analistas funcionales los requerimientos y definiciones del modelo de Negocio. Este cuerpo, que no sólo define los requerimientos de la aplicación, sino también define la injerencia que cierto requerimiento tiene para cubrir la necesidad del cliente. Esta Gerencia también coordina el soporte al sistema en Producción, relevando los incidentes detectados, definiendo si corresponde una solución, un requerimiento, o si el incidente no es un error.

D.2 Implementación del Requerimiento

El área de **Desarrollo** tiene la responsabilidad de realizar la implementación de los requerimientos relevados por la Gerencia de Producto. Es en esta *instancia* la solución debe ser implementada con las herramientas provistas por la Arquitectura respondiendo a los atributos de calidad deseados.

D.3 Quality Assurance/Quality Control

El área de **QA** Quality Assurance y **QC** Quality Control no es realizado por la misma empresa. Sino que esta actividad, es efectuada por una empresa externa a la organización.
Es esta empresa quien ha definido, en base a los requerimientos y sus condiciones de entrada, los casos y datos para realizar regresión sobre cada *release* que se despliega en producción como así también los casos de los nuevos Pedidos de Cambio. La interacción con este equipo es mediante entregables conformados por:

- *Release* de código desplegado en ambiente operativo.
- Incidentes reparados para el actual *release*.
- Pedidos de Cambios para el actual *release*.
- Requerimientos implementados en el actual *release* (incluye especificación del Req.).
- Ambiente: conjuntos de funcionalidad habilitada, y contexto (interfaces habilitadas, datos de terceros, etc.).

Dado este input y mediante una herramienta compartida este grupo reporta los **incidentes detectados** en la salida de las operaciones que se realizan sobre este *release*. En todos los casos se verifican, luego, los incidentes detectados y reparados con anterioridad para cerrar el ciclo de cada incidente.

D.4 Milestones o Hitos del proyecto

El proyecto ha sido realizado en varias etapas, en cada una de éstas se cubre más funcionalidad y es por ello que es posible brindar el servicio a más usuarios de un sistema anterior al actual.
Estos *milestones* conllevan despliegues en el ambiente de Producción de nuevo código funcional, cuyos usuarios finales utilizan luego de ser migrados. Dentro de estos *milestones* también tenemos diferentes despliegues no sólo para introducir arreglos a defectos sino también mejoras de performance entre otros.
La frecuencia entre despliegues no es fija y **depende** en gran parte del **criterio de aceptación** para cada *release* como así también el tiempo que se requiere para realizar la **regresión** sobre este *release*.
Idealmente la regresión debería cubrir toda la funcionalidad de la aplicación con énfasis en los nuevos cambios introducidos.
Cada *milestone* no solo habilita más clientes sino también más funcionalidad e interacciones con nuevos sistemas externos.

La armonía de los sistemas de *misión crítica* debe ser validada antes del despliegue en producción no solo en su funcionalidad sino también en performance.

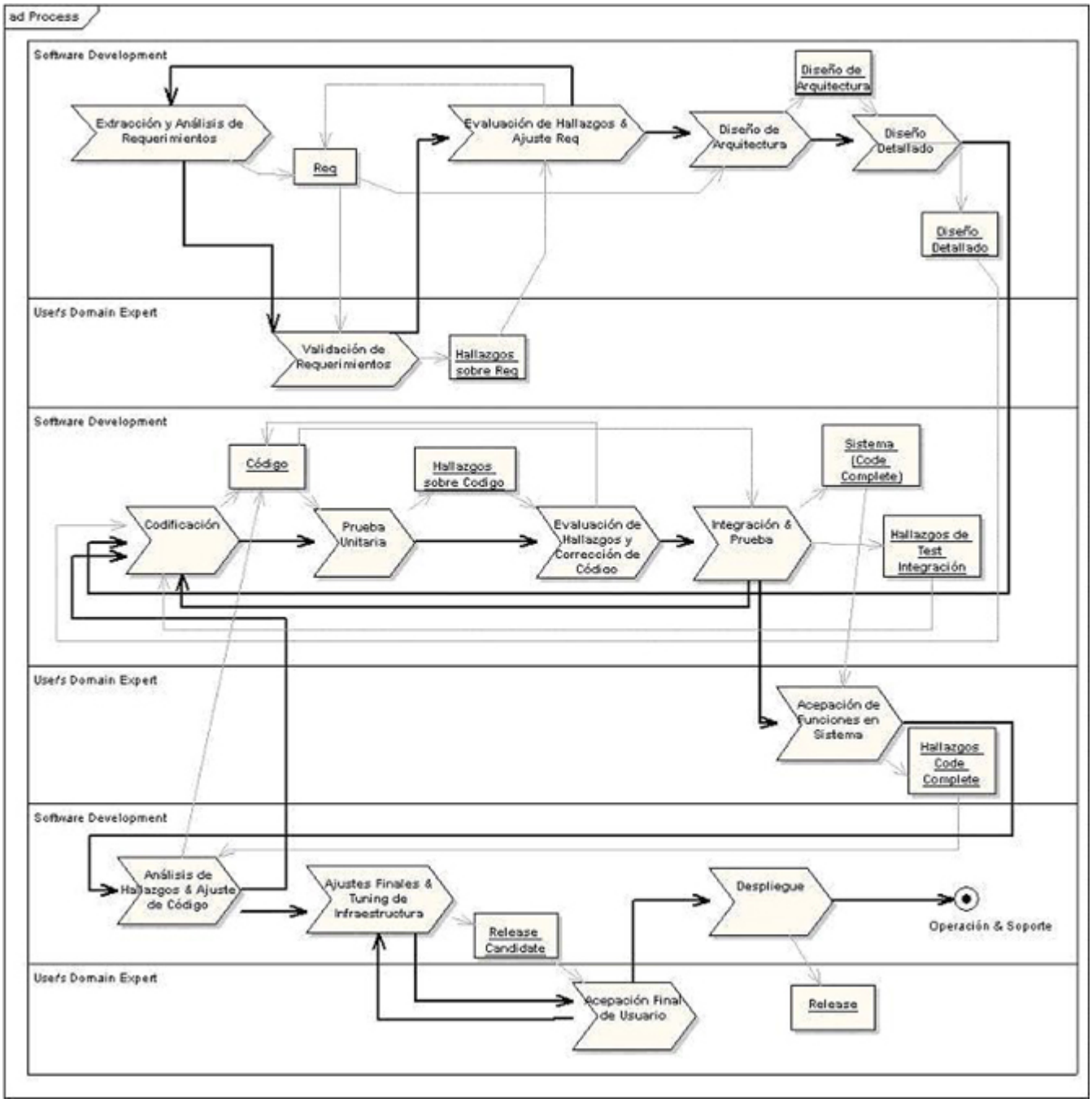

Figura 38. Ciclo de vida

D.5 Verificación del Sistema ante Cambios

Sistemáticamente y mediante *releases* el sistema que se desarrolla es verificado. Existen diversas fuentes de pedidos de cambio de un sistema, en particular:

- Incidente detectado en un ambiente productivo
- Incidente detectado por QA/QC
- Incidente detectado por Producto
- Incidente detectado por desarrollo

En todos estos casos, el incidente debe ser analizado para verificar que funcionalmente no introduce errores al modelo de negocio y el mismo actualiza la documentación pertinente.

Luego de este paso el desarrollador recibe en su lista de tareas diarias el pedido de cambio o pedido de reparación con una descripción del comportamiento funcional que debe tener el sistema. Es en este punto que el desarrollador puede acceder a la documentación para informarse sobre el impacto del cambio, pero no obstante ésto, no existe una **trazabilidad al código fuente** para comprender los **puntos de impacto** del cambio.

El desarrollador reproduce el incidente utilizando una base de datos de desarrollo, sin datos productivos. En caso de no poder reproducirlo, puede utilizar la base de datos de un ambiente QA a modo lectura para reproducir el inci-

dente.
Si el incidente o pedido de cambio no se puede recrear, luego no se puede realizar el mismo, dado que se requiere regenerar el comportamiento explícito de la aplicación para tener un control de los cambios que se introducen.
Una vez que este contrato entre los objetos que intervienen se ha explicitado, se introduce el cambio teniendo en cuenta de no destruir contratos con otros objetos.
Si como resultado de la invocación de un método se agrega un objeto a cierto *DynamicBean*, luego estamos seguros de no impactar en quienes reciben este *DynamicBean* a menos que alguno de estos objetos sobrescriba el objeto, en cuyo caso el receptor del *DynamicBean* recibiría el último objeto.

Una vez que el desarrollador ha introducido el cambio, modificado cierta lógica de cálculo u otra, éste ingresa el cambio al *CVS* [SEP2005][31] y de esta manera integra el cambio. Asimismo es responsabilidad del desarrollador documentar en el código este cambio, agregar un comentario al integrar el código al *CVS* y por último acceder a una herramienta de *tracking* de cambios para marcar el pedido de cambio como integrado.

Posteriormente, y bajo criterio del Administrador *SCM*[32] se realiza un *TAG* [SEP2005] de la herramienta de versionado CVS para generar un **baseline** [SEP2005] del sistema. Este *TAG* conformará una *release* que será desplegado en algún ambiente para su verificación.

Como input del proceso de validación se tiene una lista de los cambios introducidos por el desarrollador, pero sin un análisis de impacto, y eventualmente alguna recomendación de las áreas que se pueden ver afectadas.
Como output de este proceso, el equipo de Test consigna en la herramienta de seguimiento de errores el éxito en la verificación funcional del cambio o el incidente.
Dado este output se define si se resolverán estos incidentes reportados y se realizará un nuevo *baseline* y despliegue en test o si la versión desplegada en test será aceptada y desplegada en QA.
En general, y en este caso de estudio, se realizan tres despliegues en testing por cada despliegue que se realiza en QA.

Luego de desplegar el baseline en QA, este equipo realiza una regresión completa del sistema teniendo en cuenta las modificaciones que se han realizado en la documentación por el cambio implementado. Es allí donde se generan en el caso que corresponda, nuevos casos, y nuevos datos para la nueva funcionalidad. Aunque nuevamente el impacto de los cambios es analizado a nivel funcional, y no se tienen en cuenta las relaciones internas en las cuales impacta el cambio, excepto en los casos en que el desarrollador ha realizado algún análisis y lo ha documentado.
En la actualidad este equipo **demora tres semanas en realizar una regresión completa** antes de autorizar un despliegue en Producción. Es por ello que esta verificación es parte del **camino crítico** para desplegar en Producción e implementar soluciones para el cliente.

[31] CVS: Concurrent Versions System.
[32] *SCM*: Software Configuration Manager.

En el caso de detectar un incidente en este ambiente, y por lo complejo de realizar un nuevo despliegue, el mismo debe ser solucionado mediante un "'**parche**"', es decir un conjunto de acciones específicas sobre componentes específicos.
Debido a la complejidad del sistema, detectar un incidente en este ambiente requiere que un desarrollador obtenga el mismo *baseline* de código, reproduzca el ambiente, genere la solución, analice el impacto y por último genere los archivos binarios específicos de la solución con un análisis de impacto.

Este "'**parche**"' es aplicado en el ambiente QA, y se realiza una nueva verificación del incidente, si éste se ha solucionado, el parche es marcado en la herramienta para aplicar en Producción cuando se realice un despliegue con esa versión.
Empíricamente se puede verificar que el parche generado no había sido detectado en el ambiente de test, y que gran parte de los incidentes **se deben a errores introducidos indirectamente al realizar un cambio sobre otra área** de la aplicación.

Luego de esta regresión y con la autorización de la Gerencia de Producto, se continúa el procedimiento desplegando esa versión de código, con los parches asociados en Producción. El acceso a este *baseline* desplegado en Producción no es permitido por ningún integrante del equipo por razones de seguridad y auditoria.

Los incidentes reportados por los clientes son analizados en una versión de código exactamente igual, con los parches aplicados, y se debe reproducir el incidente antes de tratarlo como tal.

Luego de realizar un despliegue, se ha realizado un **causal analysis** [SEP2005], encontrando que gran parte de los problemas se deben al **escaso análisis realizado por el desarrollador al momento de integrar los cambios**.

Es entonces, foco de errores, el pedido de cambio. No por el cambio en sí, sino por el impacto sobre otras áreas teniendo en cuenta que ciertas condiciones pueden dificultar la visibilidad del impacto y por ello el costo de detectar el error en etapas de desarrollo, Test y QA.

D.6 Herramientas de Verificación en Desarrollo

> *Whenever you are tempted to type something into a print statement or a debugger expression, write it as an unit test instead. –Martin Fowler*

Al analizar los incidentes detectados en Producción se llego a la conclusión que el mecanismo más eficaz para controlar los mismos es el verdadero **conocimiento del desarrollador al introducir el cambio** y verificar los cambios introducidos con *Test de Unidad* representativos al cambio introducido.

Articular este conocimiento no es sencillo en un grupo que supera las 30 personas dedicadas a tiempo completo al desarrollo del *Software*. Uno de los problemas identificados para brindar este conocimiento al desarrollador es el poco

conocimiento de la aplicación y el *framework* que brinda. Esto, no solo implica una baja re-utilización de componentes, sino a los efectos de esta *tesis*, **desconocer realmente el impacto de un cambio introducido en el código**.

La alta rotación del personal, la poca documentación técnica o de obsoleta manutención, provocan que este contrato implícito no siempre sea dominado por quien introduce los cambios.

Con respecto a los *Test de Unidad*, desde su concepción el sistema ha sido desarrollado con un *framework* de *Test de Unidad*. El mismo centraliza los procesos necesarios para ejecutar ciertos *Test de Unidad* en la aplicación en forma aislada. Esto último dado que la aplicación posee controles de ingreso, centinelas de seguridad y permisos para ejecutar ciertos servicios que dificultan la ejecución de *Test de Unidad*. Más aún, no todos los *Test de Unidad* se pueden realizar si el modelo de datos no los sustenta. Es decir, en varios casos, realizar un *Test de Unidad* para un *ABM* es sencillo dado que se comienza por un alta y luego se puede asegurar la existencia para hacer bajas o modificaciones. Otras veces las reglas de **negocio complejas** dificultan la creación de estos test, dado que los datos son representativos de un modelo superior de datos especifico y no pueden ser generados aleatoriamente (identificadores de Bancos, información de cuentas coherente, etc).

No obstante este esfuerzo y este *framework*, los *Test de Unidad* no logran extenderse a las unidades más importantes de la aplicación debido a lo dificultoso de su generación y el **conocimiento** que debe tener el desarrollador sobre los **contratos que respeta cada unidad**. Más aún, en un momento de estabilización de la aplicación o mantenimiento, donde quien desarrolló originalmente la unidad ya no se encuentra en el equipo de desarrollo.

Otro problema de los *Test de Unidad* desarrollados es que los mismos no han tenido una evolución con el código, como se esperaría, por lo que muchos de ellos no son tan útiles al verificar a la unidad que representan.

Con cierto éxito se realizan algunas simulaciones mediante *robots* [SEP2005] para asegurar cierto circuito de casos de uso, pero el control sobre la salida de estos robots no es verificado. Luego sólo se tiene la certeza que el robot puedo "avanzar" sobre algún proceso funcional, pero nada se sabe sobre los cómputos que se realizan y si éstos son correctos.

Un ejemplo de estos robots es la generación de una transferencia, o la consulta de un reporte. En el primer caso que la transferencia se genere responde a una secuencia de pasos fijos, pero no puede cubrir todas las posibilidades de autorización y liberación de la misma, más aún teniendo en cuenta restricciones que dependen del contexto como autorizaciones habilitación del cliente para realizar esa transferencia entre otros.

En el caso de un reporte tampoco se puede asegurar que el mismo responde a lo que el usuario desea consultar o debería poder visualizar sino sólo que se ha realizado una generación exitosa de información.

El contexto dicta a su vez que conforme pasa el tiempo, el conocimiento del conjunto de desarrolladores como masa **disminuye**, dado que los desarrolladores originales ya no formar parte del equipo y **la generación de *Test de Unidad* no se continua**.

Los *Test de Unidad* no son actualizados y conforme pasa el tiempo estos pierden vigencia.
Esto conlleva a que ciertos test pueden resultar en **fail**, y sin embargo no ser representativos al momento de tomar la decisión de integrar la solución al código.

Salir de esta situación no es sencillo, dado que no existe el conocimiento para generar los *Test de Unidad* y tampoco es posible utilizar el conocimiento de los desarrolladores actuales. Proyectando el esfuerzo, se vería que muchos de los *Test de Unidad* podrían perder su vigencia y en el corto plazo retornar a una situación similar.

Nuestra estrategia, no solo ha sido inspirada en este caso, sino que ataca precisamente esta situación. Mediante los esfuerzos de QA se generan *Test de Unidad* automáticamente, que brindan al desarrollador esa visibilidad sobre el cambio introducido.

El *Test de Unidad* generado puede ser regenerado durante la evolución del *Software* y así colaborar a la detección temprana de errores y reducir los parches que se deben realizar en producción.

D.7 Comparación de esfuerzos y participación de los *Test de Unidad* en el desarrollo

Debido a que no existe información para comparar esfuerzos de generación automática con generación de test de unidad en el caso de inspiración anterior, presentaremos a continuación un caso de estudio con información relevante.

En este caso de estudio se cuenta con el esfuerzo requerido para realizar los *Test de Unidad*, debido a que se han realizado en su gran mayoría y adicionalmente se han realizado *Diagramas de Secuencia* de la aplicación.

Dado que disponemos de esta información, comparar el esfuerzo necesario para generar *Test de Unidad* representativos será mas fácil y nos permitirá proyectar tendencias.

El proyecto de codificación ha requerido **400PD**[33], de los cuales se ha dedicado **60PD** para la generación de *Test de Unidad*.

Los *Test de Unidad* no cubren todas las *clases*. Adicionalmente se ha dedicado **60PD** durante la construcción del *Software* para realizar *Test de Sistema* del sistema que se desarrolló.

La participación en los costos y cantidad de **PD** de los recursos *recursos desarrollador* y *recurso testing* se puede observar en la tabla I.

Es interesante que los *Test de Unidad*, perduran y son **reutilizables**, mientras que los *Test de Sistema* que se han diseñado y generado **sólo se han documentado** y **ejecutado**.

[33] P.D.: Person Days.

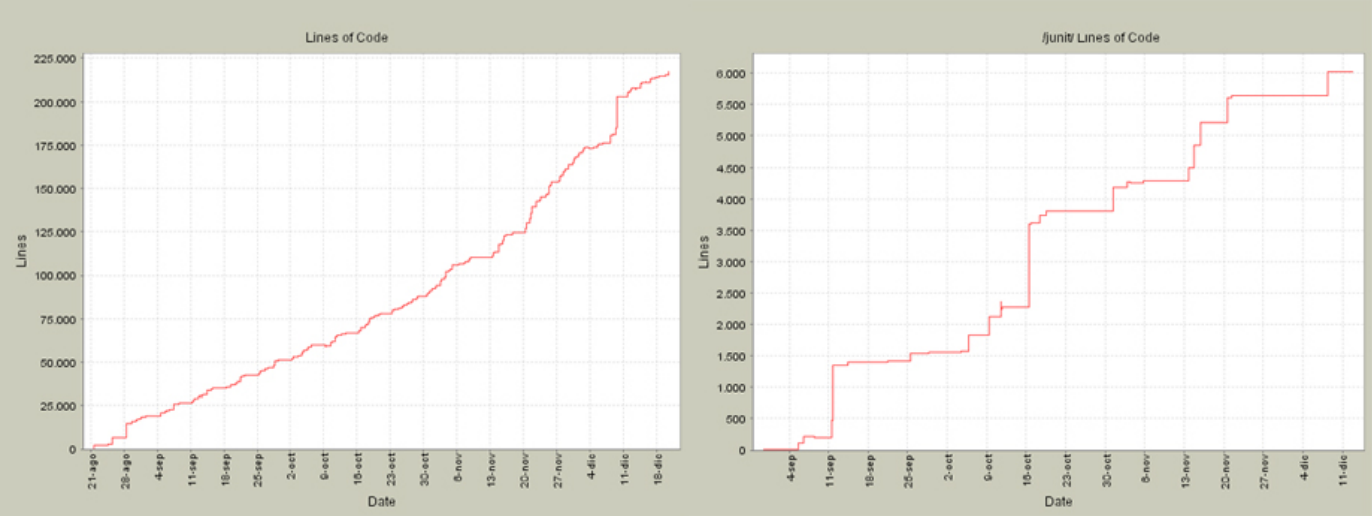

Figura 40. Comparacion entre lineas de código desarrolladas y lineas de código dedicadas a test de unidad

TABLE I
PARTICIPACIÓN EN LOS COSTOS DE GENERACIÓN DE TEST DE UNIDAD Y TEST DE SISTEMA

Tipo de Recurso	Porc. de Costo	PD (Person Days)
Desarrollador	70	60PD
Recurso Testing	30	60PD

El costo de realizar una regresión por los **111** casos de uso, requiere de no menos de **15PD**. Al generar un *release* se ejecutan automaticamente los *Test de Unidad* con una base de datos de referencia mantenida por los propios desarrolladores. Si todos los test de unidad finalizan correctamente se realiza el despliegue del *Software* y se lo somete a la regresión mediante los *Test de Sistema*.

En la Figura 40 se puede observar la evolución y participación, en líneas de código, de los *Test de Unidad* generados.

Cabe aclarar, que es una *mala práctica*, pero real, utilizar a los *Test de Unidad* como variable de ajuste en un *cronograma*. De esta forma, se reduce el cubrimiento que los *Test de Unidad* pueden realizar sobre la funcionalidad desarrollada.

Es por esta mala práctica, que muchos casos de uso no poseen *Test de Unidad* "interesantes". Esto se puede apreciar en la figura 40, donde la proporción de generación de *Test de Unidad*, (en su participación de Lineas de Código) se reduce en la etapa final del proyecto.

Sin embargo, la regresión del sistema es una tarea que constantemente se realiza y como ha sido afirmado antes, es una percepción volátil del sistema.

Hemos aplicado nuestra implementación al proyecto con una base de datos de referencia, y hemos ejecutado ciertos casos de uso, generando *ejecuciones controladas exitosas* del sistema y sus respectivos *Runs*.

Luego podemos comparar los costos de generación de *Test de Unidad* de similar magnitud, como se puede ver en la tabla II. Donde el *Costo 1* es el "Costo de generación de *Test de Unidad*" y el **Costo 2** es el "Costo de generación de *Test de Unidad* automáticamente"

TABLE II
COMPARATIVA DE TEST DE UNIDAD GENERADOS CON TEST DE UNIDAD AUTOMÁTICAMENTE GENERADOS

Caso de Uso	Costo 1	Costo 2
CU001 - Login	40 min.	5 min.
CU003 - Alta de Usuario	30 min.	4min.
CU035 - Alta de Cliente	35 min.	6min.

En cada una de las ejecuciones no sólo se capturó la información para generar test de unidad similares, sino que también se generó información que potencialmente podría generar Test de Unidad para *clases* que pertenezcan al *Test de Sistema* almacenado en la *base de Test de Sistema*.

Sin duda éste es un muy reducido ejemplo, y mayores datos deben ser generados para verificar la ventaja de *Nuestra Propuesta*, aunque como se puede observar los resultados preliminares son muy alentadores.

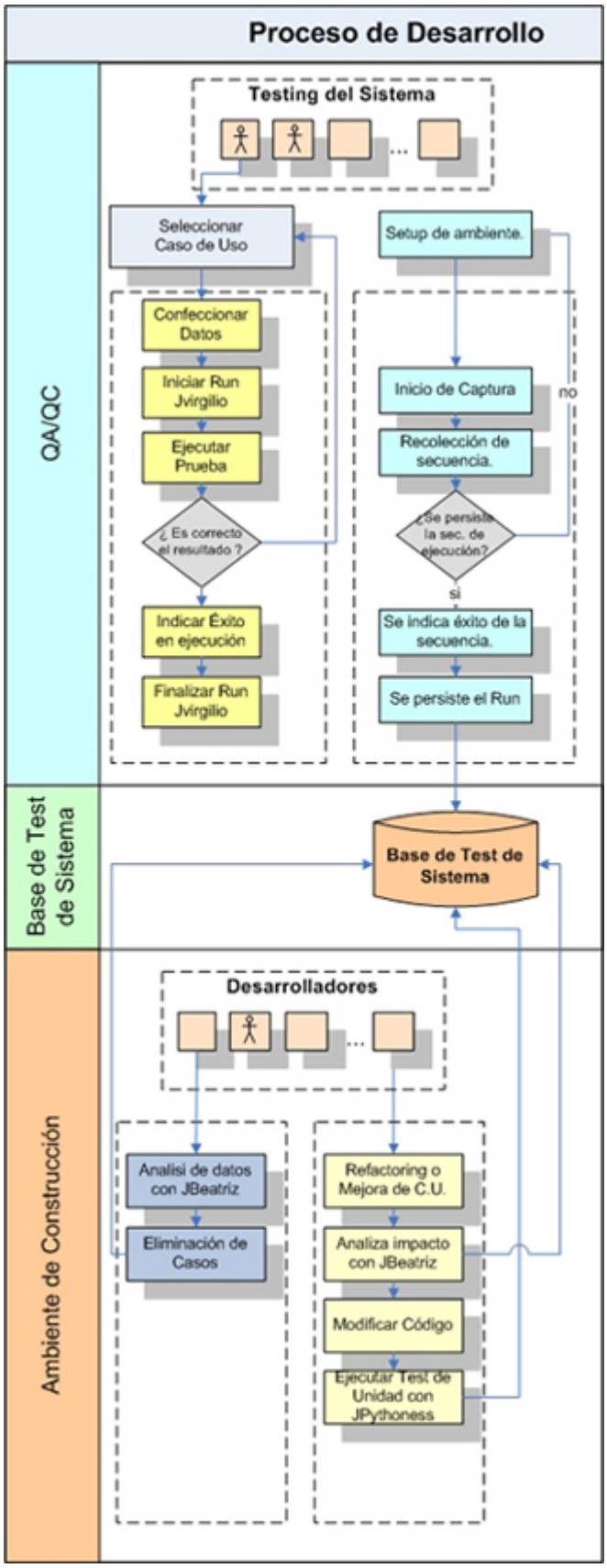

Figura 39. Diagrama simplificado del proceso de Utilización de la Herramienta

VI. Futuros trabajos

As for the Future, your task is not to foresee, but to enable it. –Antoine de Saint-Exupery (1900-1944)

A continuación desarrollaremos algunos puntos a investigar de *nuestra estrategia*, cada uno de estos representa una línea de trabajo para extender nuestra *tesis*.

A. Modelo de Datos

Dado que los casos de test generados dependen de cierto estado de un Modelo de Datos, el generar un *stub* que se comporte de la misma forma y que al mismo tiempo responda consultas distintas, no es trivial.

Los casos generados interactúan -potencialmente- con un sistema de Modelado de Datos, es decir una Base de Datos u otro dispositivo de almacenamiento de datos. Estos sistemas poseen de por sí un estado que evoluciona con el tiempo, y -potencialmente- su estado no es el mismo luego de ejecutar cierta funcionalidad.
Sin embargo *nuestra estrategia* no tiene en cuenta este cambio de estado en el Modelo de Datos y supone que la misma puede ser una "copia" del Modelo de Datos utilizado antes de la generación de los *Test de Unidad*. Si a esta copia se le aplican los *Test de Unidad* responderán correctamente a lo solicitado y si existe una diferencia en la respuesta del caso, el mismo es analizable y responde también al interés del desarrollador que verifica su cambio: ¿Es mi cambio "resistente" a estos test, o existen problemas con ciertos inputs?.

Luego el problema aquí radica en el gran estado del Modelo de Datos. Como tal, no solo debería responder a las mismas consultas que se ejecutaron en la generación del *Test de Unidad*, sino que también debería poder responder a nuevas consultas, generadas en un posible *refactory*, en el mismo contexto de estado.

El primer caso es sencillo, dado que existen hoy día mecanismos de *stub* que responden las mismas consultas. Pero en el caso de un "híbrido", es decir poder responder a las mismas consultas y nuevas consultas sobre el mismo estado, se debería desarrollar un mecanismo que permita que la consulta se realice en el mismo "estado" del Modelo de Datos.

B. Dependencia de Clases con Estados generados (fixture)

En general la dependencia de ciertas *clases* con respecto a cierto conjunto de *clases* puede analizarse estáticamente y así "descubrir" si la *instancia* de la *clase* requiere de otra *instancia* con su histórico. Sin embargo, dado que las *instancias* pueden potencialmente realizar invocaciones dinámicas, este análisis no es trivial.

Al obtener este conjunto de *clases* que no poseen una dependencia con un *fixture* [WEBMAS], podremos generar el conjunto de objetos que son plausibles de generar *Test de Unidad*.

C. Cubrimiento con Test de Unidad

La cantidad de *Test de Unidad* generada no aporta mucha información sobre la calidad de cobertura de los mismos. Es por ello que *nuestra estrategia*, en principio, no brinda información del cubrimiento de las *clases*.

Dado que los casos son generados con información resultado de la ejecución de un caso funcional, podemos decir que el caso es relevante, pero no podemos afirmar que cubre la unidad de código bajo análisis.

Este estudio de cubrimiento es pertinente dado que a partir de un conocimiento del nivel de cubrimiento podríamos ejecutar sólo los *Test de Unidad* representativos.

El análisis de cubrimiento no es trivial y puede ser analizado con diversas estrategias, a saber: cubrimiento de sentencias o instrucciones, cubrimiento de decisiones y cubrimiento de condiciones [SEP2005].

En este sentido existen distintas estrategias para lograr test de unidad representativos, es el ejemplo de *"Eclat: Automatic Generation and Classification of Test Inputs"* [EAG2005]. Este trabajo estudia un sistema bajo análisis, sus caso de *Test de Unidad*, y mediante ejecuciones "descubre" ciertos patrones de ejecución. Luego genera un subconjunto de estos *test de unidad*, que se supone revelan mejor los errores posibles de la unidad bajo análisis. Esta estrategia ha sido implementada y creemos que la integración con *nuestra estrategia* podría resultar en un conjunto de *Test de Unidad* más representativo.

D. Criterio de decisión de Test exitoso

Al recrear el estado de la *instancia* que genero el *Test de Unidad*, se logra un estado "similar" al que alguna vez se logro, luego se ejecuta el método bajo análisis y su respuesta es analizada con la respuesta almacenada.
En el caso de *rules* y servicios de infraestructura, en general el resultado es fácilmente comparable, dado que podría ser un valor de verdad *true/false*, sobre el éxito de cierta *rule*.

Sin embargo a medida que aumenta la complejidad de las estructuras de los tipos comparados, notamos que existen diversas formas de comparar el *resultado obtenido* con el *resultado almacenado*.
Una forma es comparando la estructura del resultado, es decir, comparar la *taxonomía* del mismo y así evidenciar si cumple con cierto *contrato explícito*.

En otro nivel, se podría comparar el valor específico obtenido dentro de esta *taxonomía*. Esto no sólo se reduce al método ***equals*** de los objetos, sino analizar las diferencias. Dado que podrían existir valores de referencia y un resultado ser válido para cierto valor de entrada si se encuentra en cierto rango.

No se ha abordado este aspecto en *nuestra estrategia*, sino que se brinda soporte para incluir un método de comparación de estructuras y valores.

E. Verificación empírica del beneficio

Debido a lo joven de la *estrategia* y la reducida implementación de referencia, se debe completar información para asegurar el beneficio del esfuerzo. No en todos los casos *nuestra estrategia* es la más útil y eficiente. Sin duda, en sistemas donde las interfaces evolucionan mutando sus argumentos, los *Test de Sistema* de la **base de test de sistema** no tendría una continuidad importante en el tiempo, sino que perderían vigencia rápidamente. En un ambiente donde las interfaces no varían fuertemente los *Test de Sistema* podrían tener vigencia por mas tiempo.

F. Generador de Diagramas de Secuencia

A partir de los **Run** también se podría generar *diagramas de secuencia* que representen el desempeña dinámico de cierto **Run** funcional. Esto muchas veces no es posible con herramientas de *ingeniería reversa* [SEP2005] debido a que la cooperación entre instancias es definida en forma dinámica [34].

En general, realizar estos *diagramas de secuencia* no es sencillo debido a la experiencia necesaria sobre el desarrollo de la aplicación.

Como se puede apreciar en la figura 41 el *diagrama de secuencia* presenta las interacciones entre distintas *instancias* del *sistema* para resolver algun escenario de un *caso de uso*.

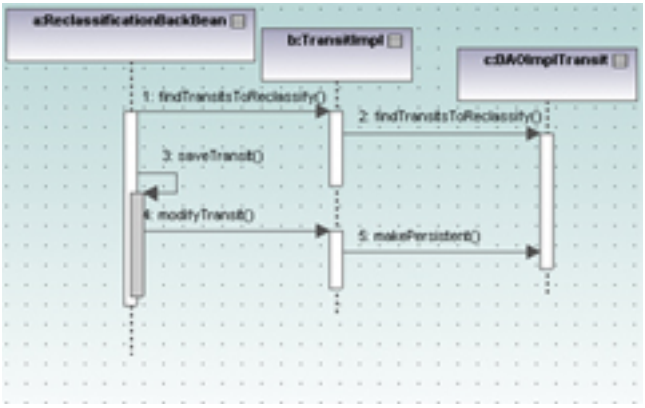

Figura 41. Diagrama de Secuencia

Si las *instancias* de estas *clases* hubieran sido capturadas en algún *Test de Sistema*, sería posible visualizar su *diagrama de secuencia*. Para ello podríamos agregar una *view* a **JBeatriz** que presente estos datos. En definitiva lo que se presentaría es el mismo *Test de Sistema* almacenado, con los tipos de los métodos invocados, y si fuese requeridos datos de contexto capturados.

En general en los *diagramas de secuencia* de alto nivel, y a los efectos de presentar la generalidad del caso de uso,

[34] Las aplicaciones de tres capas suelen poseer algún mecanismo de búsqueda de servicios dinámicamente. Bajo este contexto realizar ingeniería reversa no es tan sencillo.

se podrían quitar ciertas *clases* del conjunto de **clases observadas**. Esto generaría *diagramas de secuencia* de alto nivel y personalizados a la necesidad.

G. Comparación paso a paso del resultado obtenido con el resultado almacenado

Debido a que ciertos *Test de Unidad* pueden fallar al ejecutar el **ABA**, podríamos realizar una comparación acción a acción y darle cierto peso a cada test de unidad con respecto a como ha llegado a ejecutar el **ABA**. Si el *Test de Unidad* no se llegase a ejecutar debido a que falla al intentar *re-estimular* la *clase* entonces el peso del *Test de Unidad* podría **ser nulo** y no ser tenido en cuenta a la hora de presentar resultados al desarrollador. En el caso de avanzar acción a acción y obtener siempre el mismo valor que el *resultado almacenado* del *Test de Unidad* podría tener **peso máximo**.

Incluso la cantidad de *Test de Unidad* que fallan o tienen poco peso sería un dato de la calidad de los *Test de Sistema* almacenados en la **base de test de sistema**.

H. Selección de clases candidatas para el conjunto de clases observadas

Nuestra estrategia no es responsable de seleccionar las *clases* que serán parte del conjunto de **clases observadas**. Sin embargo *nuestra estrategia* se podría extender utilizando conceptos como *Número Ciclomático* [DCM1989] de una *clase*, para seleccionar automáticamente aquellas *clases* que resulten "interesantes" bajo este concepto.

Para cierto *sistema* con grafo $\mathcal{F}$, El *Numero Ciclomático* es

$$v(F) = a - n + 2$$

donde $\mathcal{F}$ tiene $\boldsymbol{a}$ arcos y $\boldsymbol{n}$ nodos. El número ciclomático mide el número de caminos linealmente independientes del grafo $\mathcal{F}$ que representa al *sistema*.

TABLE III
Distribución del Número ciclomático con respecto a la complejidad

Complejidad Ciclomática	Evaluación de Riesgo
1-10	Sistema simple
11-20	Sistema moderado
21-50	Sistema complejo
50 o más	Sistema inestable

A partir de la complejidad ciclomátiva de McCabe podríamos:

- Determinar el número de caminos a través de cada clase en un sistema dado.
- Predecir las *clases* o módulos que son más propensos a errores.
- Determinar el número de casos de prueba para asegurarse que todas las sentencias de una clase han sido ejecutadas al menos una vez.

Otras métricas de análisis de código podrían ser utilizadas para seleccionar automáticamente este conjunto de **clases observadas**.

I. Matriz de Trazabilidad

La *Matriz de Trazabilidad* [SEP2005] es una representación gráfica de las relaciones entre dos o más productos o *artifacts* del proceso de desarrollo de *Software*, generalmente identificadas en las intersecciones de líneas verticales y horizontales. Por ejemplo, para representar la relación entre los requisitos y el diseño de un componente del *Software*.

Como se puede observar en la figura 42 es relevante el conocer los aspectos en los que impacta un cambio sobre ciertos componentes desarrollados. En este sentido y ante un cambio en un componente es interesante el conocer los test que cubre ese cambio para verificarlo.

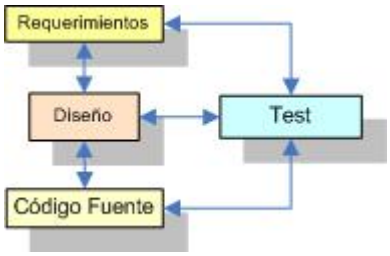

Figura 42. Diagrama de Relaciones de la Matriz de Trazabilidad

Es común que aplicaciones ya desarrolladas no tengan su *Matriz de Trazabilidad*, o que si la tienen ésta no se encuentre actualizada. En ciertos proyectos de *Software* se utilizan *tags* o *annotations* [OOS2004] de *JAVA*, por ejemplo, para documentar la utilización, sin embargo es difícil comprobar que esta información documentada sea correcta y es poco mantenible y escalable. En la figura 43, se presenta un ejemplo de documentación en el código fuente de los *casos de uso* que implementa la *clase* utilizando en un proyecto real de más de **250.000 lineas de código** desarrollado en *JAVA*.

En esta caso se ha utilizado ***@CCU 011*** para expresar que esta *clase* implementa el *caso de uso* **11** del *Sistema* desarrollado y ***@MCU 012*** para expresar que el método implementa o es utilizado en el *caso de uso* **12** del *Sistema*.

Luego una herramienta automática, recorre todas las *clases* generando justamente y con estos datos, la *Matriz de Trazabilidad*.

Nuestra estrategia se concentra justamente en esto. Es decir, dado un cambio a cierto método, generar los *Test de Unidad* que estimulan al cambio, con ciertos valores de entrada y verificar si el *Software* se comporta como se ha comportado antes. En base a conocer puntualmente esta información, sería posible realizar una *Matriz de Trazabilidad*. Y al ejecutar un cierto *caso de uso nuestra estrategia* identificaría todos las *clases* que se vería afectadas, o colaboran, en resolver este caso de uso.

Esto se lograría incorporando todas las *clases* que deseamos mantener en nuestra *Matriz de Trazabilidad* a las ***clases observadas*** y al ejecutar un cierto *caso de*

```
/**
 *
 * @author zz0khf
 * Esta clase es el Back Bean de ...
 * @CCU 011
 * @CCU 012
 * @CCU 013
 * @CCU 014
 */
 public class MatrixExampleBackBean {

  ...

    /*
    * @return
    * @MCU 011
    * @MCU 012
    */
    public String keepAll() {

  ...
```

Figura 43. Ejemplo de documentación en código de Matriz de Trazabilidad

uso se conservaran las invocaciones.

Restaría entonces, presentar la *Matriz de Trazabilidad* en forma gráfica o reportes tomando estos datos de nuestra **base de test de sistema**.

Sin duda no podríamos asegurar que sea una *Matriz de Trazabilidad* exhaustiva, a menos que hayamos cubierto todo el código con nuestras ejecuciones. Pero sin duda, la información obtenida es muy valiosa, dado que de otra forma el esfuerzo para realizarla sería importante.

J. Caducidad de los test de Sistema almacenados

Debido a que la **base de test de sistema** crece continuamente y en distintos ambientes, es importante que la depuracién de esta base de datos pueda ser automatizada para evitar generar *Test de Unidad* que no podrían ser ejecutados.

En particular, si una método cambia la signatura de su método, es decir, se modifican los tipos de datos de los parámetros, o agrega/reduce algún parámetro, o el tipo de *return* del método, entonces no sería posible utilizar los *Test de Sistema* para este método dado que ha cambiado.

Para ser más preciso, todos los *Test de Sistema* que tengan a este método en su secuencia sólo podrán ser ejecutados hasta el método inmediato anterior al método que ha modificado su signatura. Por lo tanto el resto de la secuencia del *Test de Sistema* puede ser eliminada.

Sin duda este proceso de depuración puede ser *automatizado*, y bastaría simplemente con identificar los cambios que sufren los métodos para depurar los *Run* que se vean impactados.

K. Otros temas posibles

A lo largo del desarrollo de esta *tesis* se ha analizado posibles usos de la *estrategia generada*, y aquí se explicitan estos usos que potencialmente requieren de extender la estrategia.

A partir de ciertos *test* generado se podría analizar la existencia de *código fosilizado* [SEP2005], o partes de código dentro de métodos que no son alcanzados. Esto es un *feature* del *test de caja blanca*, podría aplicarse este concepto al resultado de la exploración que se realiza.

Luego toda esta **base de test de sistema** de invocaciones de objetos podría generar contratos explícitos cuando éstos no existan y sea costoso el generarlo. Esta información es vital para comprender los valores de entrada de las unidades y poder analizar las condiciones de entrada.

Por último en sistemas *Multithreading* deberíamos distinguir entre *thread* para poder aislar **runs**.
A partir de los test generados en el modelo de datos deberíamos definir un criterio de depuración de test que no representan la evolución del código.

En general en una etapa de mantenimiento el tiempo de vida de un *Test de Unidad* es más largo, mientras que en una etapa de desarrollo los mismos poseen un tiempo de vida pequeño. Debido a que en ambientes de *misión crítica* encontramos una fuerte gestión de los *releases*, podríamos "atar" los *Test de Unidad* generados a estos *releases*, con ello los test tendrían sentido en ese contexto definido, es decir, entre *release* y *release*.

VII. Conclusiones

> *[The need to minimize software cost] suggests that large-program structure must not only be created but must also be maintained if decay is to be avoided or postponed. Planning and control of the maintenance and change process should seek to ensure the most cost-effective balance between functional and structural maintenance over the lifetime of the program. Models, methods and tools are required to facilitate achieving such balance. –Belady and Lehman, On Understanding Laws, Evolution and Conservation in the Large-Program Life Cycle, J. Systems and Software, 1(3), 1980.*

Dada la participación de los *Test de Unidad* en el proceso de desarrollo de *Software*, creemos cubrir con *nuestra estrategia* un área sumamente importante de la Industria de *Software*. En particular en los últimos tiempos varios procesos ágiles de desarrollo de *Software* han revalorizado los *Test de Unidad*, brindándole un rol protagónico.

La realidad indica, sin embargo, que el esfuerzo que se realiza muchas veces no cubre las expectativas teóricas que se ponen en los *Test de Unidad*, con lo cual no se logra el objetivo de ejercitar las unidades del sistema como se debería.

En la actualidad otras estrategias antes presentadas (Ver sub-capítulo C.1 del capítulo IV), han comenzado a surgir con diferentes variantes, pero todas concentradas en automatizar los *Test de Unidad* y así obtener los beneficios de *Tests* representativos y completos.

Nuestra estrategia, alineada con estos objetivos, ha mostrado la factibilidad de ser desarrollada, gracias a la prueba de implementación. Sin duda, experimentos de envergadura deben ser realizados para, empíricamente, obtener más indicadores del beneficio que se puede lograr. Incluso conjeturamos que los *Test de Unidad* desarrollados tradicionalmente pueden complementarse a los generados por *nuestra estrategia*. Creemos, que estos pueden ayudar a los desarrolladores a generar mejores *Test de Unidad* y explicitar los contratos de las *clases* desarrolladas.

No obstante lo joven de la implementación, podemos decir, que el camino elegido cumple con un balance que aprovecha el esfuerzo que se invierte en la verificación del *Sistema* y el costo de almacenamiento.

Nuestra estrategia aprovecha este esfuerzo para generar no solo *Test de Unidad* -que son valiosos a la hora de modificar las clases del sistema y verificar la utilización de la *clase*- sino que también brinda información de los contratos implícitos de estas *clases*.

No es sencillo cubrir todos los tipos de *sistemas*, por lo que hemos identificado los *sistemas* que se verían beneficiados por *nuestra estrategia*. Es decir, aquellos que no han explicitado los contratos que deben respetar las clases que lo componen.

Hemos visto también que los *Test de Unidad* generados mediante los *test de sistema* capturados, mantienen la particularidad de detección de errores de los *test de sistema*, brindan mejores tiempos de detección y mayor control al ingresar cambios.

Debido a la tecnología utilizada, hemos generado una implementación que analiza el *sistema* **sin modificarlo**. Consideramos este punto vital, dado que otras estrategias [ATF2006] han requerido generar su propio lenguaje para simular al sistema.

Por último esta tesis abre nuevos trabajos posibles, basándose en la **base de test de sistema** que genera. Creemos que este conocimiento del *sistema*, puede ser de importancia para desarrollar futuras herramientas que optimicen el conocimiento y brinden **visibilidad** sobre las estructuras internas del software que ha identificado *Brooks* [NSB1986].

VIII. Agradecimientos

En primer lugar, quisiera agradecerle a **Victor Braberman** por aceptar dirigir esta *tesis* y proveerme las ideas y el apoyo necesario para lograr resultados, además de aportarme su visión y propia experiencia en esta área permitiéndome continuar ante lo que parecían problemas difíciles de resolver. Por las horas de almuerzo grabadas que me brindó y por esas *telecom* desde Madrid para avanzar más raudamente.

A **Pauli**, por todas las veces que la "tesis" demoró situaciones personales, o ha requerido de mayor tiempo. Por su admirable tolerancia en estos momentos y su confianza para lograr el objetivo.

A mi **Familia** por su apoyo y preocupación constantes.

A **Marcelo Santos**, quien me acompaño en el camino de finalizar la *tesis*, incluso con su ejemplo y constancia para finalizar la suya, que contagió el deseo de cerrar una etapa de la vida.

A todos los profesionales con los cuales he compartido la estrategia y apoyaron con entusiasmo, incluso permitiendo utilizarla en sus ambientes de *QA* y *Testing*.

IX. Apéndice - MODELO DE DATOS

Ver Apéndice Adjunto.

Referencias

[LDC1321] Alighieri D.
La Divina Comedia,
Italia, 1321

[IPG1972] Belady L. y Lehman M.,
An Introduction to Program Growth Dynamics,
Statistical Computer Performance Evaluation
W. Freiburger (ed), Academic Press
New York, 1972, pag. 503-511

[CIA2005] Berg, K.van den et. al.
Crosscutting in AOSD: a conceptual framework,
Second Edition of European Interactive Workshop on Aspects in Software 2005
Bruselas, Belgica.

[MTT2002] Black R.
Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing
Capítulo 11
Testing in Context: Economics, Life Cycles, and Process Maturity,
John Wiley - Sons © 2002

[NSB1986] Brooks Jr. F.P.,
No Silver BulletEssence and Accidents of Software Engineering,
Information Processing 86. H.J. Kugler, ed. Amsterdam: Elsevier Science Publishers B.V. (North Holland):1069-1076

[OOS2004] Bruegge, B. y Dutoit A.
Object-Oriented Software Engineering,,
Using UML, Patterns, and JAVA.
Second Edition
Pearson Prentice Hall
Londres, 2004

[ASL2000] Comella-Dorda S. et al.,
A Survey of Legacy System Modernization Approaches,
Notes CMU/SEI-2000-TN-003, Abril 2000

[CDU2005] Elbaum, Sebastian et al.,
Carving Differential Unit Test Cases from System Test Cases,
DCSE, University of Nebraska - Linconl - E.E.U.U.
2005

[WEBMAS] Fowler, M.
Mocks Aren't Stubs,
http://www.martinfowler.com/articles/mocksArentStubs.html

[MRN2004] Freeman, S. et. al.
Mock Roles, not Objects,
http://www.jmock.org/oopsla2004.pdf.
OOPSLA 2004

[MAA2003] Gradecki J. y Lesiecki, N.
Mastering AspectJ: Aspect-Oriented Programming in Java
Capítulo 4
Implementing Aspect,
John Wiley & Sons © 2003

[TDD2005] Gold, R. et. al.
Test Driven Development: A J2EE Example,
Apress 2005

[UTF2004] Hamill, P.
Unit Test Frameworks,
O Reilly,
Noviembre 2004

[JTF2002] Hightower, R. y Lesiecki, N.
Java Tools for Extreme Programming: Mastering Open Source Tools Including Ant, JUnit, and Cactus
Capítulo 7
Unit Testing with JUnit,
John Wiley & Sons © 2002

[JUN2001] Kent B., Gamma, E.
JUnit,
http://www.junit.org/index.htm

[AMF2006] Kicillof N. et. al.,
Action Machines:,
A Framework for Encoding and Composing Partial Behaviors
5th Internal Conference on Quality Software
2006

[MDT1995] Leveson N.
Medical Devices: Therac-25 ,
University Of Washigton
1995

[UTIJ2003] Link J. y Frohlich, P.
Unit Testing in Java: How Tests Drive the Code,
Morgan Kaufmann Publishers © 2003

[A5F1996] Lions L. (Chairman) et. al.
ARIANE 5 Flight 501 Failure,
Report by the Inquiry Board
Paris, Julio de 1996

[STT2005] Loveland, S. et al.,
Software Testing Techniques:,
Finding the Defects that Matter
Charles River Media
2005

[DCM1989] McCabe, T. J. et al.
Design Complexity Measurement and Testing.,
Communications of the ACM 32, 12 (December 1989): 1415-1425

[TAS2004] Myers, G.
The Art of Software Testing
Second Edition
John Wiley Sons © 2004

[RTI1998] Onoma, O. et al.,
Regresion Testing in an Industrial Environment,
Comm. ACM, 41(5):81-86,
Mayo, 2005

[SCR2006] Orso, A. et al.,
Selective Capture and Replay of Program Executions,
College of Computing - Georgia Institute of Technology - E.E.U.U. 2006

[EAG2005] Pacheco C. (Masters thesis),
Eclat: Automatic Generation and Classification of Test Inputs,
MIT Department of Electrical Engineering and Computer Science (Cambridge, MA), Junio 2005.

[SEP2005] Pressman, R.
Software Engineering, A practitioner´s ppproach,
McGraw-Hill, 6ta edición.
Londres, 2005

[WEBRAE] Real Academia Española,
Diccionario de la Real Academia Española,
http://www.rae.es/

[ATF2006] Saff, D. et al.,
Automatic Test Factoring for Java,
MIT Computer Science and Artificial Intelligence Lab - E.E.U.U. 2006

[ESD1999] Subramaniam B.
Effective Software Defect Tracking
Reducing Project Costs and Enhancing Quality,
CrossTalk 1999

[PDP2003] Wassim M. y Glozic D.
PDE Does Plug-ins,
IBM Canada Ltd. September 8, 2003
http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

[TIA2001] Watkins, J.
Testing IT: An Off-the-Shelf Software Testing Process
Capítulo 5
Unit Testing,
Cambridge University Press © 2001

[TIG2006] Willem V., et al.
Test Input Generation for Java Containers Using State Matching,
NASA Ames Research Center / Masaryk University
2006

[TAA2003] Zeyu Gao J., et. al.,
Testing and Quality Assurance for Component-Based Software
Capítulo 8
Test Automation and Tools for Software Components,
Artech House © 2003

[WEBCAC] *WEB: Cactus*
http://cactus.sourceforge.net/

[WEBECL] *WEB: Eclipse*
Eclipse Development IDE
http://www.eclipse.org

[WEBSWT] *WEB: Eclipse*
SWT Development
http://www.eclipse.org/swt/

[WEBEXT] *WEB: Extreme Programming*
http://www.extremeprogramming.org/

[WEBJAV] *WEB: JAVA*,
Sun Microsystems
http://java.sun.com

[WEBDBU] *WEB: Jakarta Project - BeanUtils*
Apache
http://jakarta.apache.org/commons/beanutils/

[WEBJFA] *WEB: Eclipse*
JFace Wizards
http://www.eclipse.org/articles/Article-JFaceWizards/

[WEBJTE] *WEB: JTest*,
http://www.parasoft.com/jsp/products/

[WEBDIF] *WEB: GNU Diff for Java*
GNU Open Source
http://www.bmsi.com/java/diff

[WEBHIB] *WEB: Hibernate*,
Hibernate: Framework de persistencia de Datos
http://www.hibernate.net

[WEBMYS] *WEB: MySQL Motor de Base de Datos* ,
http://www.mysql.com

[WEBREF] *WEB: Refactoring*
Martin Fowler
http://www.refactoring.com/

[WEBXMO] *WEB: XMojo*
JMX Implementation
http://www.xmojo.com

[WEBXML] *WEB: XMLDiff*
XML Difference Analisys
http://tools.decisionsoft.com/xmldiff.html

[WEBXST] *WEB: XStream*
Frameword de serialización de Objetos.
http://www.xstream.codehaus.org

PhD. Braberman, Victor es Profesor de la Universidad de Buenos Aires, ha dictado diversas materias de Ingeniería de Software en el Departamento de Computación de la Universidad de Buenos Aires. Sus áreas de investigación principales son "Formal Verification" y "Software Analysis". *PhD. Victor Braberman* dirige un grupo de investigación en Ingeniería de Software, DEPENDEX, el cual ha publicado diversos papers en ediciones de importante referato en la Ingeniería de Software. TEL: +54 11 4576–3359, e-mail: vbraber@dc.uba.ar

De Simoni, Fernando Fernando De Simoni es Universitario en Ciencias de la Computación. Se desempeña como ayudante de 1ra regular del mismo Departamento de Computación en el área *Ingeniería de Software*. Desarrolla su carrera profesional en empresas multinacionales en la Industria de Software. Su ámbito de influencia en estas empresas se concentra en el Diseño y Arquitectura de sistemas de *misión crítica*. En particular, utilizando la tecnología JAVA/J2EE. TEL: +54 11 4 816–3731, fax: + 54 11 4816–3731, e-mail: ldesimon@dc.uba.ar

ÍNDICE

Índice de figuras