Evaluación Empírica de las Pruebas de Sistemas: Un Caso de Estudio

Juan Pablo Amador Arévalo

juan.amador@ucr.ac.cr

Marcelo Jenkins Coronas

marcelo.jenkins@ecci.ucr.ac.cr

Abstract.

La siguiente investigación busca evaluar los *system tests* realizados por los desarrolladores con el propósito de mejorar la calidad de las pruebas con respecto a los errores detectados por los clientes desde el punto de vista de un *Product Owner* en el contexto de la organización. Esto se hizo con un caso de estudio de dos diferentes versiones de un mismo producto de *Software* de una empresa desarrolladora.

Esta investigación es valiosa porque en esta empresa históricamente no hubo suficientes recursos para tener un departamento de QA, lo que aumenta la responsabilidad a los desarrolladores de crear y probar el código. Es más valioso identificar los errores que se cometen actualmente para generar mejores pruebas y por ende mejorar la calidad del producto.

El caso de estudio se realizó mediante la elaboración de reportes de cobertura de los pruebas realizadas en cada uno de los *releases* de julio 2013 y enero 2014 los que permiten contraponerlos contra los errores y arreglos que se hicieron en estos mismos *releases* y así caracterizar la causa de los errores y cómo se pudieron haber evitado dichos errores.

1 Introducción

1.1 Justificación

La literatura relativa a *testing* siempre hace hincapié en que un desarrollador no debería de hacer las pruebas del sistema, a lo sumo debería realizar las pruebas unitarias del código que desarrolla [1]. Sin embargo, en ciertas organizaciones por falta de madurez o falta de dinero, existe la ausencia de un Departamento de QA por lo tanto los mismos desarrolladores realizan también el proceso de pruebas.

La empresa de este caso de estudio es una *startup* y como tal mucho de su desarrollo hasta el momento ha sido hasta cierta forma empírico. La organización sigue siendo muy inmadura en muchos de sus aspectos por lo que en sus primeros 2 años mucho del desarrollo era muy desordenado y sin un Departamento de QA.

Previendo este problema, se creó un *Domain Specific Language* (DSL) definido sobre Java que permitiera de una manera fácil y entendible realizar *system testing* sobre la aplicación, esto con la idea de motivar a los desarrolladores a realizar sus propias pruebas y que las mismas queden automatizadas para permitir realizar refactorizaciones con algún grado de garantía de que el código sigue funcionando como se espera.

Actualmente, ya existe un departamento de QA, sin embargo, aún no es suficientemente maduro y se sigue dependiendo de los desarrolladores para que realicen la mayoría de los *system tests* de la aplicación. Incluso se ha decidido que los DSLs aun van a ser realizados únicamente por los desarrolladores, pero poco a poco el departamento de QA realizará las tareas de *testing* y el desarrollador nada más tiene que programar estos escenarios en los DSLs.

1.2 Objetivos de investigación

El objetivo de la presente investigación es analizar los *system tests* realizados por los desarrolladores con el propósito de mejorar la calidad de las pruebas con respecto a los *bugs* detectados por los clientes desde el punto de vista del *Product Owner* en el contexto de la organización.

1.3 Contexto de la investigación

La investigación fue realizada por un solo investigador analizando el código y los errores detectados de la organización. Se realizó el análisis de las pruebas existentes y los errores que ocurrieron en dos *releases:* el primer *release* se hizo en Julio del 2013 y otro que se realizó en enero del 2014; esto para considerar la madurez que pudieron haber tenido los desarrolladores haciendo *System tests* y poder observar cómo ha evolucionado la cobertura con el tiempo. La cobertura de pruebas en el código tiene dos métricas, coberturas de líneas de código, que busca medir la cantidad de líneas de código cubiertas por al menos una prueba, y la cobertura de branches, que busca ejercitar todas las posibles condiciones que tienen los puntos de decisión del código. Por ejemplo, si un "if" contiene una expresión con 3 elementos, cada uno de estos 3 elementos sería un potencial branch a ejecutar. Estas dos medidas de cobertura se analizaron en la presente investigación.

2 Marco teórico

2.1 DSL

DSL significa *Domain Specific Language*, y lo que pretende es crear un pseudo-lenguaje de programación para alguna tarea o tema en particular. Este pseudo-lenguaje se puede montar sobre cualquier lenguaje de programación existente, por ejemplo Java. Lo que se hace con el DSL es de cierta forma enriquecer el lenguaje existente con nuevos comandos que solo aplican en el contexto del DSL.

Existen muchos estudios que hablan de usar DSL para extraer información del dominio y al mismo tiempo esconderle al usuario y a otros desarrolladores la solución técnica de la implementación. En [2] mencionan claramente esta característica del DSL, y al mismo tiempo en [3] se realiza una revisión de literatura de las ventajas y desventajas de los DSL y la gran variedad de sistemas que usan este tipo de lenguajes para diversas funciones.

Fowler en [2] menciona dos razones por las que los DSL se han vuelto muy populares: mejoran la productividad de los desarrolladores y mejoran la comunicación con los expertos en el dominio.

No existe una manera standard de crear un DSL. Hay patrones que se recomiendan seguir para la creación del mismo, pero lo más importante es que el código DSL tiene que ser claro y simple, y esto no para los ojos de un programador, sino para alguien que puede ser ajeno a la programación pero que conoce mucho del tema en cuestión.

La construcción de un DSL puede ser muy laboriosa ya que cada uno de los comandos que se proveen al usuario puede esconder una lógica muy compleja; sin embargo si se plantea bien, los beneficios pueden ser mayores al costo de crearlo.

Por lo general, los DSL están compuestos de verbos y sustantivos, los sustantivos denotan contextos sobre los cuales trabajar, en otras palabras, cada sustantivo tiene un set de verbos que se le pueden aplicar. Un mismo verbo puede que no sea aplicable a dos sustantivos diferentes. Por lo mismo, a la hora de construir un programa en DSL, se van definiendo sustantivos que permiten aplicarle verbos para que este sustantivo realice acciones. Esta capacidad de extraer el dominio de un problema lo hace una muy buena herramienta de *testing*, y por eso justamente se usó en la organización con este fin.

En el contexto de la organización en estudio, los DSL se usan como herramienta de *testing*. La idea fue crear un lenguaje basado en comandos que permitiera describir escenarios de una forma muy sencilla. Los comandos tienen sentido solo en el contexto de la organización, pero permite a cualquier persona que conozca del negocio crear un *test case* en cuestión de minutos, aunque no conozca nada de programación.

Por el otro lado, sobre *system testing* existe una gran cantidad de literatura pero como se resume muy bien en [4] toda la literatura converge en que este tipo de pruebas tienen que ser realizados por agentes externos al desarrollo, algo distinto a lo que se realiza en la organización en estudio. Por lo anterior es que este caso de estudio es novedoso en los resultados que pueda generar

Por ejemplo, la figura 1 muestra un ejemplo de un DSL.

Fig. 1. DSL de Ejemplo

```
application()
    .user()
        .login()
        .doorMotor()
            .addDeviceToUser()
                .repeat()
                    .updateStatus(MotorState.UP)
                    .sleep(5, TimeUnit.SECONDS)
```

```
                .updateStatus(MotorState.DOWN)
            .until(10, TimeUnit.MINUTES);
```

Este DSL es para generar un ambiente y probar el motor de un portón. Esta prueba va a probar el funcionamiento de un motor, cambiando su estado de abierto a cerrado durante 10 minutos y ver como la aplicación reacciona a esta funcionalidad. Adicional a esto se puede ver como se prueban diferentes elementos de la aplicación como el autenticar un usuario y agregar un dispositivo a dicho usuario.

En la figura 1 podemos ver que existen comandos que en ningún otro contexto tendrían sentido, sin embargo en el contexto del programa anterior, nos permite definir de una forma muy clara y legible, una serie de acciones de un usuario sobre un motor para un portón. El comando `updateStatus` solo existe dentro del sustantivo `doorMotor`, ya que solo un dispositivo tiene estatus que actualizar. De igual manera existen comandos genéricos como el `repeat`, que son accesibles desde cualquier contexto ya que son comandos auxiliares para diversas funcionalidades.

2.2 System tests

Los *system tests* son las pruebas encargadas de verificar la funcionalidad del sistema verificando los objetivos del programa que se definen en la documentación del usuario. [1]. En otras palabras, los *system tests* son los encargados de probar la aplicación tal y como la va a usar el usuario. A diferencia de las pruebas de integración, los *system test* no tienen visibilidad sobre el código o estructura de la aplicación, sino que son pruebas completamente de caja negra. Ni siquiera se basan en especificaciones externas porque buscan probar discrepancias entre la especificación inicial y la forma en que se está usando la aplicación. [1]

Es altamente recomendado que este tipo de *testing* no lo realicen los desarrolladores de la aplicación, sino que debería de ser un departamento de QA independiente encargado el que lleva a cabo este *testing*. Incluso, idealmente, debería de ser un departamento de QA externo que no tenga relación con la empresa que realizó el software, esto para evitar conflictos o polarización de opiniones.

Existen muchos tipos o niveles de *system tests* que se pueden llevar a cabo dependiendo del objetivo que se quiere validar, pero para lo que refiere a este caso de estudio, sólo nos interesa el concepto de *system tests* como un todo, la noción de validar que el sistema realiza lo que el usuario espera.

3 Diseño del estudio empírico

3.1 Preguntas de investigación

Las preguntas de investigación fueron:

1. ¿Cuál es la efectividad de los *System Tests* generados por los desarrolladores?
2. ¿Qué errores cometen los desarrolladores al realizar los *System tests*? (Omiten escenarios, omiten requerimientos, los *tests* no prueban lo esperado)

3.2 Tipo de Estudio y Sujetos de Análisis

Se realizó un caso de estudio sobre el proceso de desarrollo de la organización. Esta organización es un start-up de alrededor de 60 personas que desarrolla un sistema para proveer fácil entrada al negocio de Internet of Things a empresas. El negocio de esta empresa es el desarrollo de chips que se integran con prácticamente cualquier dispositivo, estos chips van a reportar los estados de las diferentes cualidades de este dispositivo a una nube también programada por la misma organización. De igual manera este chip provee la capacidad de mandarle comandos a los dispositivos, lo que permite controlar y monitorear remotamente diferentes dispositivos domésticos.

3.3 Procedimiento de recolección de datos

Se realizó un reporte de cobertura de todos los *tests* de DSL (*System tests*) que tenía la aplicación en dos *releases* distintos, uno a mediados del 2013 y otros a comienzos del 2014; esto para poder sacar un promedio y no penalizar lo que pudo haber sido un mal *release*. Para esto se utilizó el plugin de Cobertura de Java que permite generar reportes de cobertura sobre cualquier tipo de pruebas, no solo JUnits. Se recopiló esta información para saber que líneas están cubiertas por *tests* y cuales líneas no.

Usando la herramienta de *error tracking* Jira, que es la que usa la empresa, se sacaron reportes de todos los errores (no mejoras) reportados en estos dos *releases*. Manualmente se analizó el repositorio de manejo de versiones (GIT) y se buscaron los *commits* asociados a la solución de estos problemas (la herramienta muestra cada *commit* hecho para resolver un caso) y así se obtuvo las líneas de código modificadas para cada problema.

En esta empresa existe una muy estricta política que no se puede hacer un arreglo en el código si no existe un Jira que refleje la necesidad del cambio, esto nos garantizó que los datos eran válidos.

La diferenciación entre errores y mejoras de software es muy importante: error se refiere a que el cliente esperaba una funcionalidad y no la está cumpliendo o no la está cumpliendo bien, una mejora es que el cliente pidió una funcionalidad extra o una extensión a la existente. Este segundo caso obviamente no se puede prever con *tests*.

3.4 Procedimiento de análisis de datos

Por cada error reportado en Jira se contrapuso el reporte de cobertura que se generó con los *tests* existentes contra el reporte de archivos y líneas modificadas para ese Jira del repositorio de manejo de versiones, para analizar si las líneas que se modificaron para arreglar los problemas tenían cobertura o no. Posteriormente se clasificó la causa del bug en una de cuatro categorías:

1. No existía un *test case* para esa funcionalidad
2. Existía un *test case*, las línea estaba cubierta por el *test case*, pero no cubría todos los escenarios.
3. Existía un *test c*ase pero no cubría la línea con el error.

4. No se podía hacer un test

Con esto fue que se realizó el estudio para determinar cuál es la mayor causante de problemas, si la ausencia de *test* cases (tipo 1), si *test* cases incompletos (tipo 2), o si *test* case mal hechos (tipo 3). También se sacaron datos adicionales como si la cantidad de (tipo 4) es muy grande, sería una señal de que la herramienta necesita ser aumentada para contemplar todos los elementos que no pueden ser probados en este momento.

3.5 Procedimiento de validación de datos

El proceso de validación fue manual, sobre todo en el caso que el error haya sido catalogado como tipo 3 en el análisis. Hay que validar que el *test case* existente si tuviera como objetivo probar la funcionalidad que tuvo el error, y no que fuera un *test case* para otra funcionalidad y que en realidad ese error se tenía que haber calificado como tipo 1.

Todos los tipo 1 hubo que validarlos para ver porque no existía ese *test case*. Las razones pueden ser: que el desarrollador no haya pensado en ese escenario en que podía ser utilizado el producto, o que el caso de uso que generó el error estaba mal documentado y por ende no era esperado que fuera un caso que el usuario ejecutara.

4 Descripción de resultados

4.1 Descripción del estudio y los sujetos escogidos

La tabla uno nos muestra un resumen de las dos versiones de la aplicación en estudio, se puede ver el tamaño de la aplicación, el lenguaje de programación de ambas aplicaciones como la fecha en que fueron creadas.

Table 1. Versiones en estudio

Versión	Fecha	Número de líneas	Lenguaje
1.2	Julio 2003	236,069	Java
1.8	Enero 2014	301,373	Java

El estudio como ha sido explicado anteriormente se basa en el análisis de dos versiones de software de la organización en estudio. Estas dos versiones son la 1.2, la cual salió en Julio del 2013 y la 1.8 la cual fue liberada en Enero del 2014. Estas dos versiones se seleccionaron porque marcan dos tiempos importantes de la empresa. La versión 1.2 fue la última versión estable que se realizó antes de tener un proceso de desarrollo ágil y más ordenado y formal. Esta versión duro mucho tiempo en el campo y alcanzó un gran porcentaje de cobertura entre los clientes. También fue la última versión que funcionaba solo con una base de datos relacional y sin posibilidad de escalar horizontalmente. Por el otro lado, la versión 1.8 ya utiliza una base de datos no relacional para las funcionalidades básicas del sistema, además de proveer facilidades para escalar horizontalmente. Las versiones de 1.3 a 1.7 fueron los primeros intentos en esta dirección

(escalar horizontalmente y usar una base de datos no relacional), sin embargo tuvieron errores, o funcionalidades incompletas que impidieron que se pudieran liberar al público de forma oficial. La versión 1.8 también tiene la particularidad de haber sido usada por más clientes, ya que a todo cliente nuevo se le instalaba esta versión.

Entre 1.2 y 1.8 existen casi 8 meses de desarrollo, 8 meses en que la cultura organizacional sufrió cambios y la cantidad de desarrolladores creció de gran medida, todos estos cambios hacen de estas dos versiones, sujetos muy interesantes para analizar y estudiar para ver los cambios que hubo entre uno y el otro, además de complementarse para ver cómo se han comportado las diferentes versiones del *software* de la organización con el tiempo.

4.2 Cobertura del estudio

El estudio cubre dos versiones de la solución de software que produce la organización en estudio, los cuales reflejan un año de trabajo, entre desarrollo y mantenimiento. Estas dos versiones permiten sacar una pequeña radiografía del proceso de desarrollo de pruebas y un poco del desarrollo de la empresa en general y a su vez da señales de puntos específicos a mejorar.

5 Hallazgos

El caso de estudio generó varias conclusiones interesantes, pero vamos a empezar contestando las dos preguntas de investigación.

En las siguientes tablas se encuentra un resumen del caso de estudio realizado. En la tabla 1 se puede ver un resumen de los datos de cobertura para las dos versiones en estudio. Primero, podemos ver los datos de cuantas líneas existen en total y cuantas no están cubiertas por las pruebas de DSL. En la siguiente columna podemos ver el porcentaje que esto significa. En las últimas dos columnas podemos ver los mismos datos pero relativos a los *branches* que tiene el código.

Table 2. Resumen de los resultados de cobertura

	Instrucciones no cubiertas	Cobertura de líneas	Branches fallidos	Cobertura de branches
1.2	164,309 de 236,069	30%	10,173 de 12,563	19%
1.8	161,189 de 301,373	47%	11,585 de 15,478	25%

En la tabla 2 podemos ver el resumen de todos los errores reportados en la herramienta Jira para estas dos versiones en estudio. Los errores en estudio son los que están bajo la columna de errores reales, sin embargo, llama la atención que existe una buena proporción de los errores reportados que terminaron no siendo errores o no siendo arre-

glados por una u otra razón, lo que habla de que puede existir un problema con el proceso de reporte de errores, ya que están llegado una gran cantidad de falsos positivos hasta los desarrolladores.

Table 3. Resumen de errores reportados

	Errores reportados	Errores reales	No se reprodujeron	Duplicados	Not a Bug	Won't fix	Errores con el ambiente de instalación
1.2	36	20	1	8	3	1	3
1.8	76	44	5	6	11	0	10
Total	**112**	**64**	**6**	**14**	**14**	**1**	**13**

Por último en la tabla 3 podemos ver la categorización de los errores en las cuatro categorías definidas antes. Similar a la tabla 2 los datos están separados por cada uno de las dos versiones en estudio, así como un total que resume los resultados del caso de estudio.

Table 4. Resumen resultados

	No se puede hacer test	No existía test	Existía test y la línea estaba cubierta	Existía test y la línea no estaba cubierta
1.2	5	9	4	2
1.8	15	14	5	10
Total	**20**	**23**	**9**	**12**

5.1 RQ1. ¿Cuál es la efectividad de los System Tests generados por los desarrolladores?

Parece ser buena, y comparando la versión de código 1.2 y la versión 1.8 va en fuerte aumento, y esto tiene sentido pues entre estas dos versiones se hizo un gran esfuerzo en la creación de *system tests* con el DSL para todas las nuevas funcionalidades e incluso para funcionalidades existentes. En la versión 1.8 se alcanzó un 47% de cobertura de líneas y 25% de branches, este resultado deja cierto sin sabor porque un problema que se detectó en el sistema es la existencia de mucho código muerto, una práctica que se ha estado haciendo en esa organización desde el comienzo y es que cuando se hace una mejora de algún desarrollo o se cambia alguna funcionalidad, el código anterior no se borra, se deja ahí. Esto afecta directamente el reporte de cobertura, porque puede que parte de ese 53% de líneas que no estén cubiertas sean líneas muertas, sea código que ya no pueda ser ejercitado, por lo que en realidad se tiene una mejor cobertura de la que se está demostrando.

Esta limitante no se puede sobrepasar hasta que se haga el esfuerzo de limpiar todo el código muerto de la solución de software y así poder tener una idea clara del tamaño del software y la cobertura real.

5.2 RQ2. ¿Qué errores cometen los desarrolladores al realizar los *system tests*?

Las dos mayores causas de errores fueron porque los ingenieros no realizaron pruebas para algunas funcionalidades. Hilando más fino, la organización en estudio tiene diferentes equipos de desarrollo en varias ubicaciones geográficas, y es curioso determinar que de los 14 errores que se dieron en la versión de software 1.8 por no hacer las pruebas con los DSL, 8 fueron de un mismo equipo. Esto indica que la cultura de utilizar el DSL para todo desarrollo está encontrando cierta resistencia en este equipo en particular, y es algo que hay que trabajar. Otro número alto que se encontró y que ciertamente es una alarma, es que muchos de los problemas encontrados que no tenían una prueba de DSL asociado, fue porque la herramienta de DSL no permite hacer *tests* para estos casos, esto tiene que revisarse y analizar si no vale la pena extender la herramienta para soportar todos estos casos.

Entrando en los casos en que si existía un test, sobretodo en la versión 1.8, el mayor error fue que si se hizo el *test case* para la funcionalidad pero se omitió alguno de los requerimientos. De nuevo este escenario hay que evaluarlo, porque si un desarrollador no hizo la prueba para algún requerimiento y aparte se está encontrando un error en este requerimiento, puede que no haya entendido todo el requerimiento cuando se estaba programando la funcionalidad, y esto si hay que evaluarlo y trabajarlo para que no suceda porque puede generar errores graves.

Por último también hubo casos en que se hizo el test, pero incompleto, se probó el requerimiento, pero el test case no se pensó de forma tal que ejecutara todos los caminos de error. Se esperaba que este número fuera mayor ya que las personas que están generando los *test cases* no son ingenieros de QA, sin embargo, este número fue el menor de todas las causas con un total de 9 errores en las dos versiones por esto.

Otra conclusión a las que podemos llegar como resultado de este caso de estudio es que existe una gran cantidad de errores que están llegando hasta el desarrollador y que en realidad no eran errores. De los 112 errores reportados para estas dos versiones, solo 64 terminando siendo errores y requiriendo un arreglo de un desarrollador, los otros 48 pudieron haber sido resueltos por las etapas previas de *troubleshooting*. Que un 43% de los errores reportados se descarten porque en realidad no eran errores es un dato alarmante que también tiene que tratar de corregirse.

5.3 Evaluación de la validez del estudio

Uno de los problemas fue relacionado al análisis de los datos, ya que una de las preguntas de investigación era si los *System Tests* generados por los desarrolladores eran efectivos. Para esta tarea se evaluó la cobertura que estas pruebas generaban. Sin embargo, una vez realizados los reportes de cobertura para las dos versiones, se detectó un problema importante. La aplicación en estudio tiene mucho código muerto, sobretodo en la versión 1.8, por lo tanto, por más que la cobertura subió de un 30% a un 47%, existe la sospecha de que la cobertura de líneas de código efectivas es aún mayor. Esto debido a que en la versión 1.8 se cambió mucho código, pero siempre dejando la versión original en la aplicación. Por lo tanto por más que los resultados no hablan de una buena

cobertura generada por los *system tests* definidos por los DSL, queda la sensación de que la realidad es aún más alentadora de lo que presentan los números.

Otro problema es que la relación entre los cambios de código y la cobertura no es trivial. Para un arreglo de un problema en el programa, se pueden realizar muchos cambios de código, cada uno de estos cambios puede tener una situación de cobertura diferente, por lo tanto se necesita de un criterio experto para determinar cuál fue la razón por la que no se descubrió este error con un *test*. En el caso de este estudio, el criterio experto en la mayoría de los casos fue el mío, para tratar de disminuir el riesgo a sesgo, algunos de los problemas analizados, se discutieron con otros integrantes del equipo en la empresa donde se realizó el caso de estudio para llegar a la clasificación y así evitar que todas las clasificaciones fueran dadas por la misma persona.

Existe un potencial problema de sesgo, es que todo el análisis fue hecho con base en un filtro de Jira (el repositorio de manejo de errores), para filtrar solo los errores presentados en esa versión. Cualquier error que fuera clasificado de mala manera, como mejora en vez de bug por ejemplo, o no tuviera la versión correctamente asignada, se pudo haber perdido. Este riesgo no es tan grande por qué Jira es lo que se usa para coordinar los *Sprints* (se usa una metodología ágil en el desarrollo del sistema) y también para los reportes de calidad y productividad, por lo que se espera que los datos en Jira sean bastante precisos y confiables.

Por último dos limitantes del estudio son en primer lugar basarse nada más en la cobertura para realizar muchas de las conclusiones, la cobertura por más que es una buena métrica y sirve como base, sin embargo, tiene la desventaja ya mencionada antes, de ser drásticamente afectada por el código muerto de la aplicación.

Por último, la segunda limitante fue basarse simplemente en dos versiones del software. Sería interesante extenderlo a la versión 1.11 del producto, que es la siguiente versión estable y así poder analizar cómo siguen las tendencias que ya se marcaron en estas dos versiones.

6 Conclusiones

El estudio ha llegado a reafirmar algunas cosas que se sospechaban. Antes no se realizaban suficientes pruebas, lo que estaba causando que ciertos errores los estuviera detectando el cliente. La cultura de la empresa poco a poco ha cambiado y ya se han hecho una mayor cantidad de pruebas, sin embargo, la herramienta para realizar las pruebas (DSL) no ha evolucionado de mano del sistema y está empezando a mostrar limitantes que impiden probar ciertos aspectos del sistema, esto indica que hay una gran cantidad de casos de uso que hay que probar manualmente. También se comprobó que los módulos del sistema que no se pueden probar con DSL tienen una mayor tendencia a tener errores, de nuevo, estos módulos necesitan por completo de una prueba manual, esta conclusión puede llevarnos a inferir que el DSL si está ayudando a la calidad del producto de *software* ya que buen porcentaje de los errores no tenían DSL.

Los resultados de este estudio han tenido un impacto bastante positivo en la organización por varias razones. Primero, antes de este estudio, no existía una manera de saber el nivel de cobertura de los diferentes módulos en lo que se refiere a *System Tests*. Este

caso de estudio va a dejar este proceso como uno de sus resultados. Por el otro lado, nunca se había hecho un análisis minucioso de los errores encontrados y las posibles razones por las cuales estos errores no fueron detectados más temprano en el proceso. Este análisis ya fue compartido con ciertas personas de la organización, y se planea expandir aún más para demostrar la necesidad de expandir la herramienta de DSL e incluso demostrar que los módulos que no tienen DSL están llegando con muchos errores al cliente, lo que demuestra un problema en el proceso manual de pruebas.

7 Trabajo futuro

Sería interesante volver a realizar este ejercicio con la versión 1.11 del producto, e incluso seguir haciéndolo de una forma periódica para todas las versiones del producto, esto para ver las tendencias de crecimiento en la cobertura y para ver si los errores cometidos en una versión de software se están corrigiendo para los siguientes.

De igual forma el reporte de cobertura debería incorporarse al sistema de integración continua para que los ingenieros puedan corroborar si las pruebas que realizaron como parte del arreglo a un error o como parte de una nueva funcionalidad, en realidad está cubriendo con toda la cobertura de los cambios realizados.

Otra tarea pendiente que deja claro este estudio es la necesidad de limpiar el código muerto en la aplicación, mientras esté ahí no hay forma de saber el tamaño real de la aplicación y por ende no hay forma de medir la cobertura de la aplicación, y más bien se está llegando al punto en que nuevas implementaciones, por más que estén totalmente cubiertos bajen la cobertura ya que la nueva implementación puede involucrar menos código que la original.

8 Referencias

[1] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale y D. C. Schmidt, «Improving domain-specific language reuse with software product line techniques,» *Software, IEEE*, vol. 26, nº 4, pp. 47-53, 2009.

[2] A. Van Deursen, P. Klint y J. Visser, «Domain-Specific Languages: An Annotated Bibliography.,» *Sigplan Notices*, vol. 35, nº 6, pp. 26-36, 2000.

[3] G. J. Myers, C. Sandler y T. Badgett, The art of software testing, John Wiley \& Sons, 2011.

[4] C. Ibsen y J. Anstey, Camel in action, Manning Publications Co., 2010.

[5] M. Fowler, Domain-specific languages, Pearson Education, 2010.

[6] P. Brereton, B. Kitchenham, D. Budgen y Z. Li, «Using a protocol template for case study planning,» de *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering. University of Bari, Italy*, 2008.