Predicción del cambio a través de la historia del sistema

Juan F. Hurtado[1], Franco Sabadini[1] Santiago Vidal[2,3], and Claudia Marcos[2,4]

[1] Facultad de Ciencias Exactas, UNICEN, Tandil, Argentina,
{hurtado.juanf, fsabadi}@gmail.com
[2] ISISTAN Research Institute, UNICEN, Tandil, Argentina,
{svidal, cmarcos}@exa.unicen.edu.ar
[3] CONICET, Concejo Nacional de Investigaciones Científicas y Técnicas, Argentina
[4] CIC, Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, Argentina

Resumen El mantenimiento de aplicaciones es una de las tareas mas costosas dentro del desarrollo software. Por esta razón es importante la realización de mantenimiento preventivo que permita incorporar con menos esfuerzo futuros requerimientos disminuyendo el costo del desarrollo. En esta línea, este trabajo propone un enfoque que permita predecir las clases de un sistema que tienen mayor probabilidad de cambiar en el futuro. De esta forma, podría dirigirse el esfuerzo de mantenimiento a un subconjunto reducido de las clases de un sistema.

Keywords: Mantenimiento de software, Ingeniería reversa, Programación orientada a objetos

1 Introducción

El desarrollo de un sistema no termina cuando es entregado, sino que continua a lo largo del tiempo de vida del mismo. Por un lado, se deben corregir los errores que el mismo pueda tener, ya sean funcionales o no funcionales. Y por otro lado, cambios por parte de las reglas del negocio y de las necesidades y demandas de los usuarios pueden generar nuevos requerimientos que deben ser analizados y, en caso de ser aceptados, agregados al sistema existente. De esta manera, los sistemas de software evolucionan continuamente para afrontar los cambiantes requerimientos de software [11]. La evolución del software es importante porque las organizaciones invierten grandes cantidades de dinero en su software y dependen del mismo. Estos sistemas son una parte crítica de la empresa y es necesario mantenerlos actualizados para que sigan siendo útiles [10].

El mantenimiento de sistemas es considerado una de las tareas más tediosas y costosas en el desarrollo de software [1]. Algunas investigaciones sugieren que aproximadamente el 60% de los costos del software están relacionados a la evolución del mismo [7]. Para poder reducir costos, los cambios en el software se deben hacer de la forma más eficiente posible, afectando a la mínima cantidad de componentes y previendo futuras modificaciones. Si son conocidas

las futuras modificaciones se pueden realizar restructuraciones preventivas en el código fuente con el objetivo de hacer mas sencilla la incorporación de los nuevos requerimientos.

Con este objetivo, este trabajo busca predecir qué clases, de un sistema orientado a objetos, tienen más probabilidades de cambiar en el futuro. Se analiza la historia del sistema, la cual contiene información útil que puede ser usada para detectar qué clases han sufrido más cambios durante su desarrollo, y para tener una imagen general de la evolución de los mismos [6]. Para lograr este fin, se utiliza analiza la volatilidad del tamaño de las clases utilizando un algoritmo de volatilidad usado en el ámbito financiero. La hipótesis que sigue nuestro trabajo es que aquellas clases que han sufrido muchas modificaciones durante su historia se seguirán modificando.

El resto de este trabajo se estructura de la siguiente manera. En la Sección 2, se describen los principales problemas del mantenimiento de aplicaciones. Luego, en la Sección 3, se analiza el uso del concepto de volatilidad financiera para predecir cambios en sistemas de software. En la Sección 4, se describe el enfoque utilizado para predecir las clases con mayor probabilidad de cambio. En la Sección 5, se presentan los casos de estudio. En la Sección 6, se detallan los resultados de los casos de estudio. En la Sección 7, se presentan los trabajos relacionados. Finalmente, en la Sección 8, se presentan las conclusiones y trabajos futuros identificados.

2 Mantenimiento de aplicaciones

La optimización de un sistema, ya sea para mejorar la performance, utilización del espacio, etc., puede llegar a cambiar la estructura del mismo. Esto hace que el entendimiento del código se vuelva cada vez más complejo a lo largo de su evolución. Cuando un sistema ha sido modificado en reiteradas ocasiones, durante un periodo largo de tiempo, su estructura tiende a degradarse, produciendo sistemas difíciles de entender y modificar. Además, su documentación puede ser inconsistente o, incluso, inexistente [13].

Si la arquitectura actual no soporta el cambio a aplicarse, primero, se debería localizar el/los componente/s afectados por la modificación, y luego reestructurar los mismos [2]. De esta manera, es posible realizar reingeniería en los componentes involucrados y no en todo el sistema. Es siempre recomendable hacer un análisis del impacto del cambio, para saber cuán costosa será la modificación a introducir y qué componentes se verán involucrados. Una vez detectados los mismos, la implementación del cambio puede consistir de varios pasos en las que se observará su propagación, es decir, qué otros componentes se verán afectados, lo cual implicará cambios adicionales [2].

La historia de un sistema contiene información útil que puede ser usada para aplicar reingeniería a la versión más reciente y para tener una imagen general de la evolución del mismo [6]. Realizando un análisis retrospectivo [4], estudiando los cambios que se han aplicado para obtener las nuevas versiones del software, es posible reconstruir el proceso evolutivo. Al comparar la versión actual de

un sistema de software con las anteriores, se puede evaluar si los objetivos de la evolución se han alcanzado; alternativamente, el proceso de evolución puede servir para entender qué se ha cambiado y cómo han impactado esos cambios. Esto requiere analizar grandes cantidades de datos.

Manipular gran cantidad de información puede llegar a ser costoso, tedioso e inmanejable, y es uno de los mayores problemas en la investigación de la evolución del software, así como el análisis de varias versiones de un mismo sistema [9]. Cuando se quiere aplicar reingeniería a un sistema muy grande (de cientos, o incluso miles de clases, con una gran cantidad de versiones cada una), se observa de inmediato que el análisis de dicho volumen de datos no puede ser tratado de forma manual. Estrategias para poder hacer un análisis de cómo ha evolucionado el sistema es necesario para poder realizar las modificaciones necesarias a un bajo costo.

3 Análisis de la volatilidad de las clases a través de un algoritmo financiero

En este trabajo se propone el uso del concepto financiero de volatilidad para poder analizar las variaciones en el código. En los mercados financieros se observa los eventos históricos, con el fin de buscar su causa. Una buena comprensión de los acontecimientos en el pasado es la clave para entender mejor los mercados en el futuro [17].

Existen varios algoritmos que se utilizan para predecir el movimiento dentro de los mercados financieros. Sin embargo, no todos son aplicables a la predicción de cambios en los sistemas de software. En este trabajo se propone el uso de volatilidad ya que presenta una manera simple y precisa de predicción. A continuación se describe brevemente el concepto de volatilidad en el contexto económico y su relación en la ingeniería de software.

3.1 Volatilidad

La volatilidad, en un contexto económico, es una medida de la frecuencia e intensidad de los cambios del precio de un activo, la cual se expresa típicamente en términos anuales y puede reflejarse tanto en un número absoluto, como en una fracción del valor inicial. En el mercado de valores, se utiliza para medir el cambio de los precios con respecto al tiempo y, mediante distintos modelos, obtener un pronóstico [12] que ayude a predecir si los valores subirán o bajarán. De esta forma se cuantifica el riesgo del activo.

La volatilidad anualizada σ es proporcional a la desviación estándar σ_{SD} de los retornos del activo dividida por la raíz cuadrada del período temporal de los retornos:

$$\sigma = \frac{\sigma_{SD}}{\sqrt{P}}$$

donde P es el período en años de los retornos. La volatilidad generalizada σ_T para el horizonte temporal T se expresa como:

$$\sigma_T = \sigma\sqrt{T}$$

Por ejemplo, si los retornos diarios de una acción tienen una desviación de 0.01 y hay 252 días de intercambio en un año, entonces el período temporal de los retornos es $\frac{1}{252}$ y la volatilidad anualizada es:

$$\sigma = \frac{0.01}{\sqrt{\frac{1}{252}}} = 0.1587$$

La volatilidad mensual (i.e., T = $\frac{1}{12}$ de año) sería

$$\sigma_{mes} = 0.1587\sqrt{\frac{1}{12}} = 0.0458.$$

A mayor volatilidad, mayor es el riesgo, porque las posibilidades de que suba o baje son más altas. Un activo que es considerado muy volátil tiene un precio muy cambiante, y por lo tanto se considera poco predecible. Varios estudios consideran que la volatilidad tiene un poder de predicción relativamente bueno ya que las tendencias en la volatilidad son más predecibles que las tendencias en los precios [3].

3.2 Medición del cambio

Para analizar el cambio que sufre el código a medida que el sistema es modificado, a lo largo de las diferentes revisiones, es necesario definir uno o más criterios. Esto, se puede realizar de diferentes maneras, ya sea comparando el tamaño de los archivos, fecha de modificación de los mismos, número de líneas de código, cantidad de clases, de métodos, de variables, etc. Con el objetivo de medir el cambio de una clase se analizaron tres criterios:

1. Cantidad de líneas por clase: Se considera que una clase cambió cada vez que se agrega o se remueve una o más líneas de la misma.
2. Cantidad de métodos por clase: Se considera que una clase cambió cada vez que se agrega o se remueve uno o más métodos de la misma.
3. Complejidad ciclomática por clase: Se considera que una clase cambió cada vez que la complejidad ciclomática de la clase cambia.

El primer criterio es el más exhaustivo, dado que analiza el cambio a nivel de líneas de código, sin embargo, puede llegar a obviar cambios que se hagan en una misma línea, o incluso, el caso donde se agreguen y eliminen la misma cantidad de líneas a la clase en una revisión. El segundo es similar al primero, pero, al solo tener en cuenta cambios en la cantidad de métodos, puede llegar a perder cambios dentro de los mismos. Por último, el tercero analiza cambios relacionados a la performance de la clase, pero puede resultar el menos exacto, ya que es poco frecuente que la complejidad ciclomática de una clase se vea expuesta a cambios significantes.

Para medir el cambio en la historia de una clase, a lo largo de las diferentes revisiones, medimos la diferencia de acuerdo a alguno estos criterios. Por ejemplo, considere un sistema con dos clases (ClaseA, ClaseB), tres revisiones (Rev1, Rev2

	Rev1	***Rev2***	***Rev3***
ClaseA	100	130	135
ClaseB	87	87	70

Tabla 1. Cantidad de lineas en cada revisión de las clases *ClaseA* y *ClaseB*

y Rev3) y en cada revisión se tiene la cantidad de líneas de código de cada clase que se muestra en la Tabla 1.

Si se tiene en cuenta el criterio de cantidad de líneas de código, se comparan la cantidad de líneas entre revisiones adyacentes, y se obtiene un panorama general del cambio que sufrió cada clase a lo largo del tiempo. Para el ejemplo dado, el cálculo de cambio se muestra en la Tabla 2:

	Rev1-Rev2	***Rev2-Rev3***
ClaseA	|100-130|=30	|130-135|=5
ClaseB	|87-87|=0	|87-70|=17

Tabla 2. Cálculos del cambio en las clases *ClaseA* y *ClaseB* durante toda la historia del sistema

Como se puede apreciar en la Tabla 2, se resta la cantidad de líneas entre revisiones adyacentes y se aplica el valor absoluto del resultado (ya que solo importa la magnitud del cambio). Para este caso, el resultado muestra que ClaseA cambió en todas las revisiones, mientras que ClaseB sólo cambió en la última revisión (con respecto a la revisión original).

3.3 Adaptación del enfoque financiero a sistemas de software

En esta sección se mostrará de que forma se realizó la adaptación del enfoque desde un punto financiero a uno de sistemas de software, y cómo se adaptaron las diferentes partes. En la Tabla 3 se muestra dicha relación.

Enfoque financiero	**Sistemas de software**
Activos	Clases
Tiempo	Números de revisiones
Precios	Valor de cambio según un criterio

Tabla 3. Comparación de representaciones del enfoque financiero y de sistemas de software

A continuación, se explica en más detalle por qué se eligieron dichas representaciones:

- Los activos son las clases del sistema de software, ya que cada una se analiza por separado para calcular su volatilidad, y así poder comparar los resultados obtenidos y encontrar aquellas que son mas propensas a cambiar en el futuro.
- El tiempo es reflejado como los números de revisiones, ésta es la forma más significativa y sencilla en la que se puede medir el cambio de un sistema de software a lo largo del tiempo. Usar una unidad de medida de tiempo (como por ejemplo horas, días, semanas) no resultaría eficiente, ya que un sistema puede pasar un largo tiempo sin ser modificado, o puede recibir varios cambios en el mismo día, incluso en la misma hora, a cargo de diferentes personas. También, el acercamiento de usar las revisiones como medida de tiempo, facilita la obtención de la información necesaria, puesto que no se necesita hacer ningún calculo extra para obtener el cambio que se realizó en un momento específico.
- Los precios son interpretados por valores de diferentes criterios de cambio e indican los valores asociados a una clase en un punto en el tiempo (es decir, en una revisión específica).

En la Figura 1 se muestra un gráfico de línea con un ejemplo del cambio de dos clases (llamadas A y B) a lo largo de un sub-rango de revisiones, utilizando como criterio de cambio la cantidad de sus líneas de código.

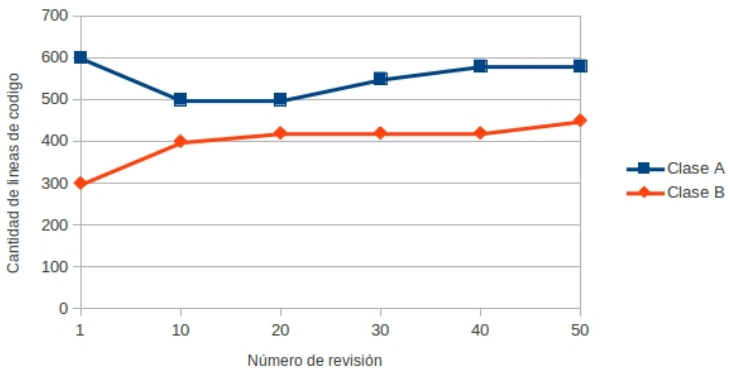

Figura 1. Ejemplo de representación gráfica del cambio de dos clases de un sistema

4 Predicción de la evolución del sistema aplicando volatilidad financiera

Analizando las clases que más varían en un sistema de software, el enfoque que se presenta en este trabajo, pretende identificar aquellas clases que tienen mayor probabilidad de cambiar en el futuro. Los cálculos predictivos se realizan con ayuda de un algoritmo financiero de volatilidad, el cual se aplica sobre la historia

del sistema. Los pasos a seguir para predecir qué clases son más propensas a cambiar en el futuro, dada la historia de un sistema, son:

1. Obtener la historia del sistema que se quiere analizar, ya sea desde un repositorio o por cualquier otro medio.
2. Obtener la información necesaria para el análisis.
3. Procesar los datos de la historia y predecir qué clases son las candidatas a cambiar en el futuro cercano.

En las siguientes subsecciones, se detallará cada paso.

4.1 Obtener historia de un sistema

Se decidió trabajar con aplicaciones orientadas a objetos y escritas en Java, debido a la forma en la que están modularizadas. Esto permite analizar una clase de forma aislada y facilitar el análisis de los datos.

Para obtener una cantidad de muestras suficientes, se decidió trabajar con las revisiones de los sistemas. Una revisión es un cambio realizado al código fuente, administrado por un CVS (Concurrent Versions System). La mayoría de los sistemas poseen un gran número de revisiones. Si se desea analizar todas, llevaría mucho tiempo y no significaría un aporte de información importante, ya que muchas de las mismas podrían ser cambios pequeños. Para acotar la cantidad de revisiones a analizar, se decidió utilizar rangos mensuales. De esta forma se toma la última revisión de cada mes.

4.2 Obtener información

Con el objetivo de analizar las revisiones se utilizo Moose[5]. Moose permite representar el código fuente de una aplicación mediante un metamodelo. Luego, pueden realizarse consultas sobre el metamodelo y calcular diferentes métricas. Por lo general, los modelos creados a partir de las revisiones de un sistema poseen mucha información, de la cual sólo una parte es necesaria para el análisis propuesto. Por esto, es importante extraer solo la información relevante desechando todo lo que no sea necesario. Específicamente, de cada clase en cada revisión se recupera su nombre completo (incluido el nombre paquete, para diferenciar clases con el mismo nombre que pudieran aparecer en diferentes lugares), la cantidad de líneas, la cantidad de métodos y la complejidad ciclomática.

4.3 Predecir el cambio

Para predecir el cambio se utiliza el concepto de volatilidad financiera. A continuación, se muestra de qué forma se analiza la información de un sistema, y se predice qué clases son candidatas a cambiar en las próximas revisiones.

[5] http://www.moosetechnology.org/

1. Se inicia la predicción, utilizando toda la información necesaria del sistema que se esta analizando, y un valor numérico N, que indica la cantidad de clases que se desea obtener como resultado (por ejemplo, si N = 4, se obtendrán las 4 clases con mayor volatilidad del sistema).
2. Se obtienen todas las versiones del sistema que fueron cargadas para el análisis, y por cada una de ellas, desde la segunda a la última, se realizan las siguientes tareas:
   (a) Se obtiene la información de todas las clases que aparecen en un rango de versiones, desde la primera de la lista, hasta la que se está analizando actualmente.
   (b) Se calcula la volatilidad de las clases pertenecientes al rango, o ventana, que se esta procesando.
   (c) Se obtienen las N clases con mayor volatilidad, y por cada una de ellas se realizan las siguientes tareas:
      i. Se busca en la lista de clases más volátiles L, si ya existe la clase que se esta analizando actualmente.
      ii. Si existe la clase en la lista, se suma 1 al valor guardado para la misma, indicando que aparece como candidata en una ventana, o rango, más.
      iii. Si no se encuentra aún en la lista, se agrega la clase y el valor 1, indicando que aparece, por el momento, como candidata en solo una de las ventanas. Luego se agrega dicho objeto a la lista de clases candidatas a ser las más volátiles.
3. Una vez que se analizaron todos los rangos, se ordena L de forma descendente, según su cantidad de apariciones en las ventanas, y se devuelven las N primeras clases de dicha lista.

Una vez finalizado el análisis, se obtiene una lista con las N clases más volátiles del sistema, para ese preciso momento. La intuición del enfoque es que los desarrolladores centren los esfuerzos de reingeniería, o cualquier otra técnica que se quiera usar, sobre dichas clases, y no invertir tiempo en otras que tienen menos probabilidad de cambiar en el futuro.

5 Caso de estudio

Todas las pruebas fueron realizadas con el código fuente de las siguiente cinco aplicaciones. Estas fueron seleccionadas, debido a que están desarrollados en Java, poseen licencias de código abierto, son de dominios diferentes, están almacenadas en SVN y poseen una cantidad de revisiones adecuada para el análisis.

- SQL Squirrel: Aplicación gráfica que permite ver la estructura de una base de datos, navegar sus tablas, enviar comandos SQL, etc., a través de JDBC1.
- JFreechart: Librería desarrollada por completo en Java, que facilita a desarrolladores exhibir gráficos con calidad profesional en sus aplicaciones.
- JHotDraw: Framework desarrollado en Java, utilizado para la creación de gráficos técnicos y estructurados.

- TuxGuitar: Editor de tablaturas de guitarra, que permite leer y escribir archivos de Guitar Pro2 .
- VerveineJ: Aplicación que permite extraer información del código fuente de sistemas Java, y exportarla para el uso en la plataforma Moose.

Cada uno de los sistemas fueron analizados con los tres criterios de cambio: cantidad de líneas de código, cantidad de métodos y complejidad ciclomática. Adicionalmente se usaron diferentes combinaciones de valores de los siguientes parámetros de prueba:

- Ventana de muestra: Dado el número de revisión desde el cual se analiza la predicción, este valor indica la cantidad de revisiones anteriores, desde dicha revisión, que se usan como muestra para los cálculos.
- Ventana de resultado: Dado un número de revisión desde el cual se analiza la predicción, este valor indica la cantidad de revisiones futuras, desde dicha revisión, que se usa para comprobar los aciertos del resultado.
- Cantidad de clases candidatas: Número que indica la cantidad de clases que se utiliza como posibles candidatas para la comprobación del grado de acierto del algoritmo.

Estos parámetros, fueron utilizados para probar el algoritmo de predicción desde diferentes puntos de vista. La ventana de muestra, se utiliza para analizar si es necesario el uso de muchas revisiones pasadas a la hora de predecir el futuro o si con solo las más recientes es suficiente. La ventana de resultado, indica qué tan pronto se comprueba la predicción (si es que sucede), es decir, si las clases que se dedujo que cambiarían lo harán en el futuro cercano o lejano, así mostrando el alcance de la predicción. Por último, la cantidad de clases candidatas, asiste en deducir si el algoritmo es bueno prediciendo muchas o pocas clases que cambiaran en el futuro.

Además, se utilizaron dos formas de cálculo: Ventanas Móviles y Ventanas Acumuladas teniendo en cuenta las siguientes referencias: M =ventana de muestra, R=ventana de resultado y C =cantidad de clases candidatas.

- Ventanas móviles: En el método de ventanas móviles, se toman las primeras M revisiones, y se aplica el algoritmo de predicción a las mismas. Del resultado obtenido, se utilizan las C clases con mayor volatilidad, y se obtienen las R revisiones futuras para calcular la precisión del algoritmo para esa ventana de muestra M. Luego, se toman las M revisiones a partir de la segunda revisión, y se realizan los mismos cálculos. Una vez que se llega al final de la lista de revisiones, se calcula la media de todos los resultados de predicción obtenidos, y ese será el porcentaje de acierto del algoritmo, para los valores de los parámetros usados.
- Ventanas acumuladas: A diferencia del algoritmo anterior, en este no se utiliza un valor fijo de la ventana de muestra, sino que el tamaño de la misma va creciendo con las iteraciones. En cambio, se comienza desde las primeras dos revisiones. Se calcula la predicción con ellas, y del resultado se toman las C clases con mayor volatilidad y R revisiones futuras y se obtiene la precisión

del algoritmo. Luego, se agrega la próxima revisión al conjunto de análisis (ahora estarán en esa lista, la primera, la segunda y la tercer revisión) y se repiten los mismos pasos. Cuando se llega al final de la lista de revisiones, se calcula la media de todos los resultados obtenidos del algoritmo, obteniendo así el porcentaje de acierto promedio para el sistema.

Los sistemas fueron analizados con los criterios de cambio (cantidad de líneas, cantidad de métodos y complejidad ciclomática), con revisiones mensuales, y con los algoritmos de ventanas acumuladas y ventanas móviles. Además, se utilizaron los siguientes valores para los parámetros de ventanas y cantidad de clases candidatas:

- Ventana de muestra: Se realizaron pruebas con ventanas de muestra de 2, 5, 10 y 50 revisiones (solo para el algoritmo de ventanas móviles). Se utilizaron estos valores, para tener una muestra general de las variaciones posibles de cantidad de revisiones analizadas. No se usaron valores mayores a 50 revisiones, debido a que eso es tenido en cuenta por el algoritmo de ventanas acumuladas, el cual eventualmente analiza todas las revisiones en una misma ventana de muestra.
- Ventanas de resultado: Se usaron ventanas de resultado de 1, 2 y 5 revisiones. Estos valores representan un futuro medianamente cercano para la predicción. Por ejemplo, el grado de acierto obtenido con una ventana de resultado de valor 2 para revisiones mensuales indica el acierto mirando 2 meses en el "futuro". No se usaron valores mayores a 5, ya que representarían un futuro demasiado lejano, y buscamos predecir cambios en el futuro medianamente cercano.
- Cantidad de clases candidatas: Se tomaron 1, 3 y 10 clases candidatas. Estos valores, indican una cantidad razonable de clases de las cuales se puede predecir el cambio futuro. Utilizar cantidades más altas, haría que se pierda grado de acierto, dado que es más complejo predecir el cambio de un número grande de clases.

Todas las posibles combinaciones de los valores de los parámetros descriptos, fueron tenidas en cuenta para las pruebas. Por ejemplo, se obtuvieron resultados con el criterio de cantidad de líneas para las revisiones mensuales con ambos algoritmos de ventana, y con todas las combinaciones de tamaños de ventanas de muestra y resultado, y de cantidad de clases candidatas. Como se especificó, las pruebas se hicieron sobre las revisiones mensuales de los sistemas, es decir, se usaron las últimas revisiones de cada mes. En la Tabla 4, se muestra la cantidad de revisiones que poseía cada sistema al momento del análisis, así como también el número que se usó para los cálculos mensuales.

6 Resultados

En esta sección, se presentan los resultados, agrupados según el algoritmo de cálculo del grado de acierto utilizado (ventanas móviles o acumuladas). Para cada aplicación se detalla:

Sistema	# total de revisiones	# rev. mensuales
SQL Squirrel	5120	88
JFreechart	2272	29
JHotDraw	777	86
TuxGuitar	929	35
VerveineJ	185	16

Tabla 4. Cantidad de revisiones de cada sistema analizado

- Promedio de grado de acierto (PGA)
- Desviación estándar del promedio de grado de acierto (σ_{PGA})
- Ventana de muestra utilizada (VM)
- Ventana de resultado utilizada (VR)
- # de clases candidatas (CC)

El primer análisis muestra los mejores resultados obtenidos con cada algoritmo de calculo del grado de acierto, para todos los sistemas estudiados. A partir de esta información, se obtienen los valores de ventana de muestra, de resultado y cantidad de clases candidatas, que más se repiten en dichos resultados. De esta manera, se consigue una combinación de valores que se toma como la más adecuada para cualquier sistema. A continuación, se muestran los resultados de acierto obtenidos para esa combinación de valores de los parámetros de ventanas y cantidad de clases candidatas, y a partir de ellos se obtiene el grado de acierto total del algoritmo.

6.1 Algoritmo de ventanas móviles

Sistema	# de lineas					# de métodos					comp. ciclomática				
	PGA	σ_{PGA}	VM	VR	CC	PGA	σ_{PGA}	VM	VR	CC	PGA	σ_{PGA}	VM	VR	CC
SQL Squirrel	0,65	0,48	5	5	1	0,52	0,5	5	5	1	0,61	0,49	5	5	1
JFreechart	0,93	0,27	10	5	1	0,86	0,36	10	5	1	1	0	10	5	1
JHotDraw	0,49	0,30	5	5	10	0,34	0,36	5	5	3	0,4	0,37	5	5	3
TuxGuitar	0,75	0,44	10	5	1	0,44	0,51	5	5	1	0,8	0,41	10	5	1
VerveineJ	1	0	5	5	1	0,83	0,41	5	5	1	0,58	0,1	5	5	10

Tabla 5. Mejores resultados de grado de acierto según los diferentes criterios de cambio para revisiones mensuales y ventanas móviles

En la tabla 5, se pueden observar algunas tendencias. Por ejemplo, usar el criterio de cambio de cantidad de líneas, por lo general, da mejores resultados que los otros dos. También, la mejor ventana de resultado, de entre las probadas, parece ser la de valor 5 (es decir, que las clases que se predijo que cambiarían, lo harán en las próximas 5 revisiones, con los porcentajes de acierto descriptos), lo cual tiene sentido, dado que cuanto más en el futuro se mire, más probable es

encontrar que una clase cambio. En cuanto a los valores de ventana de muestra, los que dan mejores resultados parecen ser 5 y 10, lo cual indica que el usar valores mayores no daría la seguridad de obtener mejores resultados (ya que la ventana de 50 no aparece en la tabla). Por último, se puede observar que la cantidad de clases candidatas que mejor se desempeño es la de valor 1.

6.2 Algoritmo de ventanas acumuladas

Sistema	# de lineas					# de métodos					comp. ciclomática				
	PGA	σ_{PGA}	VM	VR	CC	PGA	σ_{PGA}	VM	VR	CC	PGA	σ_{PGA}	VM	VR	CC
SQL Squirrel	0,64	0,21	-	5	10	0,57	0,5	-	5	1	0,6	0,21	-	5	10
JFreechart	0,74	0,45	-	5	1	0,69	0,47	-	5	1	0,83	0,39	-	5	1
JHotDraw	0,22	0,42	-	5	1	0,21	0,41	-	5	1	0,22	0,42	-	5	1
TuxGuitar	0,79	0,41	-	5	1	0,4	0,26	-	5	3	0,53	0,24	-	5	10
VerveineJ	1	0	-	5	1	1	0	-	5	1	0,93	0,14	-	5	3

Tabla 6. Mejores resultados de grado de acierto según los diferentes criterios de cambio para revisiones mensuales y ventanas acumuladas

En la Tabla 6 se pueden ver tendencias similares a la de la anterior, como el criterio de cambio, la ventana de resultados e incluso la cantidad de clases candidatas. Por otro lado, los resultados obtenidos por el algoritmo de ventanas acumuladas, son mejores para algunos casos y peores en otros, con lo cual no se puede decir que uno funciona mejor que otro a la hora de calcular el grado de acierto de la predicción.

6.3 Análisis y comparación de los algoritmos

De los resultados provistos en las tablas 5 y 6, se puede deducir que la mejor combinación de parámetros de predicción, para revisiones mensuales, sería utilizar el criterio de cambio de cantidad de líneas, con ventana de muestra de valor 5, ventana de resultado de valor 5 y cantidad de clases candidatas con valor 1. En la Tabla 7, se muestran los resultados obtenidos con dichos parámetros, para todos los sistemas, utilizando los algoritmos de grado de acierto.

En la Tabla 7 se puede ver que, en la mayoría de los casos, el porcentaje de acierto del algoritmo de predicción esta por arriba del 60%, excepto para el sistema JHotDraw. Este último, da resultados bajos debido a que el sistema es reestructurado en cada versión, es decir, para cada nueva versión que es lanzada, el sistema se ve expuesto a grandes cambios de diseño e implementación. Esto se da, porque el sistema fue creado como un ejercicio para la demostración del uso de patrones de diseño. Es por esto, que JHotDraw no sigue el ciclo de diseño y desarrollo que se ve normalmente en la mayoría de los sistemas de software, y es difícil predecir los cambios que serán aplicados al mismo a medida que evoluciona.

	ventanas móviles		ventanas acumuladas	
	PGA	σ_{PGA}	PGA	σ_{PGA}
SQL Squirrel	0,65	0,48	0,49	0,5
JFreechart	0,74	0,45	0,74	0,45
JHotDraw	0,26	0,44	0,22	0,42
TuxGuitar	0,6	0,5	0,79	0,41
VerveineJ	1	0	1	0
Promedio	0,65	0,37	0,65	0,36
Promedio (sin JHotDraw)	0,75	0,36	0,75	0,34

Tabla 7. Resultados de grado de acierto para revisiones mensuales, usando el criterio de cambio de cantidad de líneas, con ventana de muestra y resultado 5 y cantidad de clases candidatas 1

En la última fila de la tabla, se muestran el grado de acierto promedio y la desviación estándar promedio, de todos los sistemas. Como se puede apreciar, para los parámetros elegidos en esta prueba, no hay una diferencia entre los diferentes algoritmos de ventana propuestos, con lo cual, cualquiera de ellos puede ser utilizado para el calculo del porcentaje de acierto. Otra de las conclusiones que se puede observar, es que el grado de acierto obtenido es de 65%. Es decir, dado que se utiliza una ventana de muestra y de resultado de 5 revisiones con una clase candidata, significa que el algoritmo de predice correctamente en el 65% de las veces que una clase cambiara en alguna de las próximas 5 revisiones del sistema cuando se analizan las últimas 5 revisiones mensuales.

Si se remueven los resultados del sistema JHotDraw, el grado de confianza se incrementa al 75%. Estos últimos resultados son más acertados, dadas las características del sistema JHotDraw explicadas anteriormente, las cuales no lo convierten en un ejemplo de análisis adecuado para este tipo de predicciones.

7 Trabajos relacionados

Varios trabajos han abordado la problemática del mantenimiento de un sistema siguiendo la hipótesis de que aquellas clases que se modificaron durante el pasado serán mas proclives a cambiar en el futuro.

Girba et al. [8], presenta un enfoque para identificar que clases de un sistema tienen mayor probabilidad de cambiar. En forma similar a nuestro enfoque este trabajo se basa en la historia del sistema para realizar las predicciones. Sin embargo, los autores consideran que hubo un cambio solamente cuando se agrega o elimina al menos un método a una clase. Por esta razón este enfoque puede obviar cambios importantes.

Tsantalis y Chatzigeorgiou [16] presentan un ranking de sugerencias de refactoring para solucionar code smells [5]. En el ranking se ubicaran primeros aquellos code smells cuyas clases hayan sido mas modificadas en el pasado. En forma similar a nuestro enfoque este trabajo utiliza volatilidad para medir el cambio en una clase. Particularmente, han empleado modelos desarrollados para prede-

cir la volatilidad futura: randomwalk, promedio histórico, alisado exponencial y media móvil exponencial ponderada.

Tsantalis y Chatzigeorgiou también presentan enfoques para identificar problemas de state-checking (el uso de sentencias *if* para determinar el comportamiento de un objeto en lugar de utilizar polimorfismo) y la refactorización de grandes métodos [14, 15]. En este caso, los autores suponen que aquellas clases que mas cambiaran son aquellas en las que se encuentran los problemas cuya refactorización involucraría un mayor numero de componentes.

8 Conclusión

Determinar que clases son mas propensas a cambiar puede ayudar a enfocar mejor el mantenimiento preventivo de los sistemas. En este trabajo hemos presentamos un enfoque basado en el concepto de volatilidad financiera que busca predecir las clases con mayor probabilidad de cambiar. Hemos obtenido en promedio un 65% de acierto en los pronósticos. Sin embargo, consideramos que deben realizar experimentaciones con un mayor numero de sistemas y utilizando diferentes rangos de revisiones (por ejemplo, revisiones semanales).

Referencias

1. Bennett, K.H.: The staged model of the software lifecycle: A new perspective on software evolution. Tech. rep., Research Institute for Software Evolution, University of Durham, United Kingdom (2002)
2. Bennett, K.H., Rajlich, V.: Software maintenance and evolution: a roadmap. In: Finkelstein, A. (ed.) ICSE - Future of SE Track. ACM (2000)
3. Connors, L.: Options Trading and Volatiligy Trading. Best of the Professional Traders Journal Series, M. Gordon Publishing Group, Incorporated (1999), http://books.google.com.ar/books?id=FAI5AQAACAAJ
4. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns - Version of 2009-09-28. Square Bracket Associates (2009)
5. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
6. Girba, T., Lanza, M.: Visualizing and characterizing the evolution of class hierarchies. Tech. rep., Software Composition Group, University of Berne, Switzerland (2004)
7. Glass, R.L.: Loyal opposition - frequently forgotten fundamental facts about software engineering. IEEE Software 18(3), 112–111 (2001)
8. Grba, T., Ducasse, S., Lanza, M.: Yesterday"s weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In: ICSM. pp. 40–49. IEEE Computer Society (2004)
9. Lanza, M., Ducasse, S.: Understanding software evolution using a combination of software visualization and software metrics. L'OBJET 8(1-2), 135–149 (2002)
10. Lehman, M.M.: Laws of software evolution revisited. In: Montangero, C. (ed.) EWSPT. Lecture Notes in Computer Science, vol. 1149. Springer (1996)
11. Mens, T., Demeyer, S.: Evolution metrics. In: Proc. Int'l Workshop on Principles of Software Evolution (IWPSE) (September 2001)

12. Poon, S.H.: A practical guide to Forecasting financial market Volatility. Wiley Finance (2005)
13. Sommerville, I.: Software Engineering 9. Pearson Education (2011)
14. Tsantalis, N., Chatzigeorgiou, A.: Identification of refactoring opportunities introducing polymorphism. Journal of Systems and Software 83(3), 391–404 (2010)
15. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software 84(10), 1757–1782 (2011)
16. Tsantalis, N., Chatzigeorgiou, A.: Ranking refactoring suggestions based on historical volatility. In: Mens, T., Kanellopoulos, Y., Winter, A. (eds.) CSMR. pp. 25–34. IEEE Computer Society (2011)
17. Ziegler, H., Jenny, M., Gruse, T., Keim, D.A.: Visual market sector analysis for financial time series data. In: IEEE VAST. IEEE (2010)