УДК 004.412; 004.413.5

ИСПОЛЬЗОВАНИЕ АВТОМАТИЗИРОВАННОЙ КЛАССИФИКАЦИИ ИЗМЕНЕНИЙ ПРОГРАММНОГО КОДА В УПРАВЛЕНИИ ПРОЦЕССОМ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Е. Г. Князев,
старший разработчик программного обеспечения
ЗАО «Транзас Технологии»
Д. Г. Шопырин,
канд. техн. наук, доцент
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Описывается метод автоматизированной классификации изменений в контексте контроля развития программного кода, основанный на статистической кластеризации метрик изменений исходного кода. Показано применение автоматизированной классификации изменений для оптимизации процесса просмотра исходного кода и автоматизации контроля изменений на ответственных стадиях разработки. Приведен способ построения отчета по параметрам процесса разработки.

Введение

Наиболее важным активом проектов по разработке программного обеспечения является исходный код системы. Большинство современных проектов хранит всю историю изменений исходного кода в специальном хранилище, которое называется системой контроля версий. Однако эта информация доступна только тем участникам проекта, которые технически подготовлены для анализа исходного кода, т. е. в основном разработчикам, в то время как над проектом работают еще и тестировщики, менеджеры и другие специалисты, которым может быть полезна информация, полученная из исходного кода в виде списков реализованной в конкретной версии функциональности, различных отчетов и т. п. Более того, анализ истории изменений исходного кода затрудняется большим объемом входной информации. В частности, хранилище исходного кода содержит массу мелких и незначительных изменений, которые осложняют поиск важных, с точки зрения анализирующего, изменений.

Автоматизированная классификация изменений в качестве вспомогательного инструмента увеличивает производительность анализирующего при выполнении задач, связанных с анализом истории программных систем. В частности, использование автоматизированной классификации позволяет отфильтровать несущественные, по мнению анализирующего, изменения системы. Разработчик или технический лидер команды разработчиков может выделить изменения, которые привели, например, к реализации новой функциональности и сосредоточиться на них.

С помощью автоматизированной классификации изменений технический лидер может автоматизировать запрет некоторых типов изменений на определенных стадиях разработки. Например, настроить инструмент автоматизированной классификации изменений так, чтобы при внесении изменения по реализации новой функциональности на стадии тестирования формировалось автоматическое уведомление о недопустимом изменении для технического лидера и автора данного изменения.

В работе приводится несколько вариантов использования автоматизированной классификации изменений исходного кода участниками проекта, не связанными непосредственно с разработкой программного обеспечения. Автоматизированная классификация изменений дает возможность тестировщику получать информацию об изменениях, в которых реализуется новая функциональность, исправляются ошибки в виде исходного кода или

текста комментария, ассоциированного с изменением. Менеджеру проекта метод классификации изменений позволит строить отчеты с распределением изменений по типам.

Таким образом, применение автоматизированной классификации изменений исходного кода способствует повышению скорости и качества просмотра изменений кода, а также предоставляет дополнительные механизмы контроля состояния процесса разработки.

Применяемый в работе метод классификации изменений базируется на кластеризации метрик изменений исходного кода алгоритмом k-средних Мак-Кина [1, 2]. Адекватность классификации подтверждается в ходе эксперимента, описанного в работе [3]. Получено значение коэффициента согласия Кохена [4], равное 0,79, которое лежит на границе значительной и превосходной степени согласованности экспертного и автоматизированного методов классификации [5].

Аспекты применения автоматизированной классификации

Автоматизированная классификация изменений может быть полезной всем членам команды разработки. Ниже подробно описаны возможные варианты использования инструмента, реализующего автоматизированную классификацию изменений программного кода всеми участниками проекта.

Применение метода разработчиком

Рядовой разработчик программного обеспечения часто сталкивается с необходимостью просмотра большого количества изменений. Например, при подключении к проекту, который уже имеет некоторую историю, или по возвращении из отпуска или командировки. В таких случаях ему приходится внимательно читать комментарий к каждому изменению, а если информации в комментарии недостаточно, то и просматривать содержимое изменения. Затраты времени на такой процесс могут быть существенными.

Автоматизированная классификация изменений избавит разработчика от необходимости погружаться в детали каждого изменения. Ему достаточно выбрать набор типов изменений, который его интересует, и просмотреть изменения, соответствующие данному типу. На рис. 1 изображена схема просмотра изменений с выбранным фильтром по типу изменения.

Автоматизированная классификация изменений пригодится разработчику для локализации ошибки, внесенной в код в определенный период времени. В этом случае разработчику будет достаточно выделить типы изменений, потенциально влияющие на выбранный модуль, и установить конкретное изменение, нарушившее работоспособность кода.

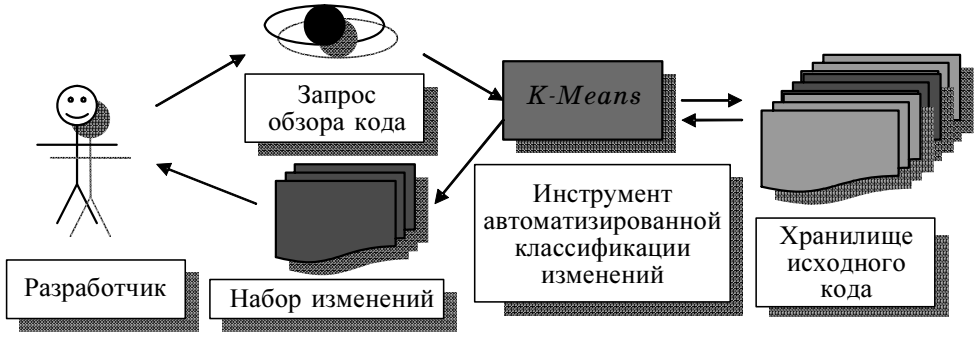
Применение метода техническим лидером

Задачами технического лидера команды разработчиков являются, во-первых, регулярный обзор исходного кода и, во-вторых, контроль изменений, вносимых на текущей стадии разработки. Обзор исходного кода — чрезвычайно полезная практика, состоящая в просмотре исходного кода на предмет поиска ошибок и проблем дизайна. Его выполнение позволяет выявить и разрешить большое количество проблем на ранней стадии разработки, пока исправление еще не требует больших затрат времени. Контроль изменений, вносимых на текущей стадии разработки, состоит в недопущении изменений, потенциально способных дестабилизировать систему на ответственной стадии процесса. По этим соображениям, например, недопустима реализация новой функциональности на финальных стадиях подготовки продукта к выпуску.

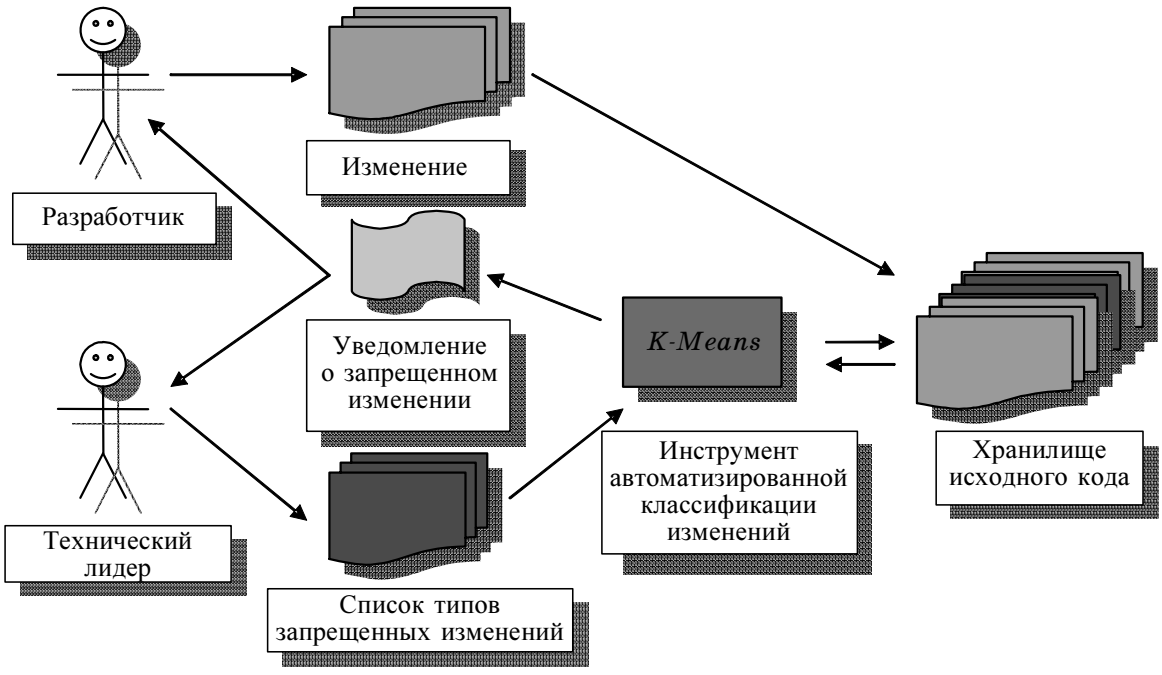
Рассмотрим задачу обзора исходного кода. Одной из основных мер поддержания качества кода на высоком уровне является постоянный просмотр

■ *Число изменений исходного кода по проектам за месяц*

Проект	Период	Число изменений
Tortoise SVN	22.09.2007–22.10.2007	215
Navi-Manager		72
KDE	17.09.2007–14.10.2007	11841 (!)



■ ***Рис. 1.*** *Просмотр изменений с фильтрацией по типу*



■ ***Рис. 2.*** *Автоматизация поиска изменений, нежелательных на текущей стадии разработки*

изменений, вносимых в код разработчиками. Команда программистов средних размеров генерирует большое количество изменений, что может приводить к физической неспособности технического лидера просмотреть все изменения.

В таблице приведены результаты подсчета количества изменений, внесенных в систему контроля версий, за период, приблизительно равный одному месяцу, для трех проектов: графического клиентского приложения для Subversion для Windows TortoiseSVN [6], клиент-серверной системы мониторинга флота Navi-Manager [7], разрабатываемой автором данной статьи в компании «Транзас Технологии», и графической оболочки для Linux и Unix KDE [8].

В некоторых проектах количество вносимых в исходный код изменений может быть очень большим. В этой ситуации технический лидер выбирает наиболее важные изменения, основываясь лишь на тексте комментариев к ним. Однако методика отбора изменений, не основанная на анализе кода, ведет к тому, что важным изменениям может быть не уделено должное внимание. Это, в свою очередь, приводит к потере контроля над качеством продукта.

Выходом из этой ситуации является использование автоматизированной классификации изменений. Просмотр кода с использованием дополнительной информации о типе каждого изменения дает возможность отсеивать неинтересные техническому лидеру изменения для более подробного изучения важных изменений.

Рассмотрим задачу контроля изменений, вносимых на текущей стадии разработки. В процессе реализации программный проект проходит несколько стадий. Например, при подготовке к выпуску версии объявляется состояние проекта `stop code`, при котором запрещается внесение любых изменений в коде, кроме исправлений найденных ошибок. Эта стадия предназначена для стабилизации версии перед выпуском.

Далее, когда все найденные ошибки исправлены, проект переводится в состояние `freeze code`, в котором разрешено внесение изменений только для исправления критичных ошибок. В качестве контроля обязателен просмотр каждого изменения еще одним членом команды перед внесением его в систему контроля версий. В этом состоянии версия исходного кода проекта находится в течение всего времени поддержки ее для заказчика.

Каждая стадия ограничивает процесс разработки определенным набором действий. В частности, в течение стадий `stop code` и `freeze code` от разработчиков ожидается деятельность по исправлению ошибок в коде, а не реализации новых функций.

Автоматизированная классификация изменений позволяет автоматизировать процесс контроля внесения изменений на текущей стадии разработки. Достаточно предоставить информацию о допустимых на текущей стадии разработки изменениях. На рис. 2 показана возможность работы модуля по обнаружению нежелательных изменений.

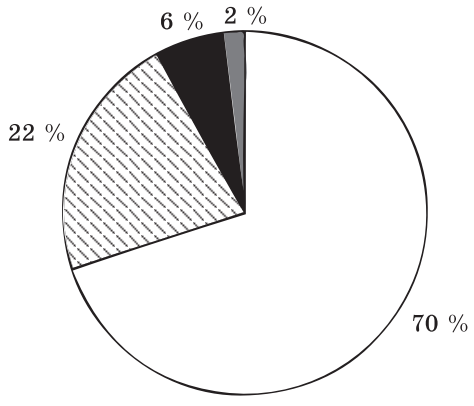
Применение метода лидером по тестированию

В процессе работы над проектом специалистам по тестированию приходится взаимодействовать с разработчиками для уточнения состояния про-

екта. Тестировщикам часто не хватает информации относительно новой функциональности, реализованной в очередной версии программного обеспечения. Иногда единственный достоверный способ выяснить полный список новых функций в конкретной версии программного обеспечения состоит в полном просмотре изменений кода за интересующий период. В этой ситуации применение автоматизированной классификации уменьшает время на проведение операции за счет отсеивания изменений, классифицирующихся как нечто кроме реализации новой функциональности.

Применение метода менеджером проекта

Менеджер проекта заинтересован в высокоуровневых показателях процесса работы. Информация о том, какая часть изменений производится в рамках реализации новой функциональности, по сравнению с рефакторингом и исправлением ошибок, позволит оценить эффективность работы над проектом. На рис. 3 показано распределение изменений по типам для проекта Navi-Manager за один месяц стадии реализации новой функциональности. Можно сделать вывод, что проект Navi-Manager движется недостаточно быстро из-за того, что основные ресурсы команды разработчиков тратятся в основном на исправление ошибок, а не на реализацию основной функциональности.



■ ***Рис. 3.*** *Распределение изменений по типам*:
□ — *правки ошибок + мелкие изменения*;
▨ — *небольшие функции + рефакторинг*;
■ — *крупные сложные функции*;
■ — *удаление кода*

Классификация изменений исходного кода

В настоящей работе предлагается использовать метод классификации изменений программного кода, который автоматизирует процесс разделения семантически различных изменений на основе значений метрик исходного кода. В качестве примеров можно привести следующие семантические типы изменений (далее — типы изменений): реализация новой функциональности, рефакторинг [9], исправление ошибки, косметическое изменение. Существует несколько методов классификации изменений программного кода. Среди них можно выделить следующие группы [10]:

— неформальные методы: автоматизированная классификация изменений посредством анализа комментариев [11, 12], метод поиска и определения типа рефакторинга [13];

— методы анализа синтаксиса изменений: эвристическое сравнение синтаксических деревьев версий [14] и анализ разницы версий при помощи встраиваемых в исходный код тегов [15].

Метод автоматизированной классификации изменений [3], описываемый в настоящей работе, основан на кластеризации значений метрик изменений исходного кода при помощи метода МакКина [1, 2]. В результате его работы производится разбиение множества изменений на заданное число кластеров, каждый из которых соответствует определенному типу изменений.

Формализация задачи

Изменение кода программной системы δ трактуется в работе как отображение множества исходных данных S в другое множество модифицированных данных S^*:

$$\delta: S \xrightarrow{\delta} S^*.$$

В некоторых современных системах хранения исходного кода каждому состоянию исходного кода последовательно сопоставляется неотрицательное целое число r, которое называется ревизией или версией программного кода. Поэтому каждое изменение исходных данных можно описать следующим образом:

$$\delta_r: S_r \xrightarrow{\delta_r} S_{r+1}.$$

Каждое изменение δ_r может быть отнесено экспертом к некоторому множеству типов изменений t_i, где $t_i \in T$ — тип изменения δ_r. При этом T представляет собой множество типов изменений, специфичное для каждого конкретного проекта. Состав множества T определяется экспертом в зависимости от специфики проекта. Во множество T обычно входят такие типы изменений, как реализация новой функциональности, рефакторинг, исправление ошибки и т. д.

Задача отнесения изменения к тому или иному типу изменений трудоемка и требует высокой квалификации эксперта, так как нет четких критериев оценки типа изменения. Введем функцию интерпретации изменений I, отображающую множество изменений $\{\delta_r\}$ во множество их типов $\{t_i\}$:

$$I: \{\delta_r\} \xrightarrow{I} \{t_1, t_2, \ldots, t_n\}.$$

В данном методе предлагается автоматизировать процесс выделения типов изменений при помощи кластеризации метрик изменений. В процессе кластеризации строится множество кластеров изменений C такое, что каждое изменение δ_r относится к некоторому кластеру $c_j \in C$.

Введем функцию автоматизированной классификации изменений I_A, отображающую множество изменений $\{\delta_r\}$ во множество их типов $\{t_i\}$:

$$I_A:\{\delta_r\} \xrightarrow{I_A} \{t_1, t_2, \ldots, t_n\}.$$

Здесь функция автоматизированной классификации I_A есть композиция функций кластеризации I_C и интерпретации кластеров I_T:

$$I_A = I_C \circ I_T, \quad I_C:\{\delta_r\} \xrightarrow{I_C} \{c_1, c_2, \ldots, c_m\},$$

$$I_T:\{c_1, c_2, \ldots, c_m\} \xrightarrow{I_T} \{t_1, t_2, \ldots, t_n\}.$$

Функция кластеризации I_C отображает множество изменений в множество кластеров. Функция интерпретации кластеров I_T отображает множество кластеров C в множество типов изменений T.

Функция кластеризации I_C может быть построена с помощью метода кластеризации МакКина. Кластеризацию изменений будем осуществлять на основе некоторых метрик изменений $M'\delta_r$. Определим понятие метрики изменения через понятие метрики программного обеспечения.

Метрика программного обеспечения (software metric) — это мера M, позволяющая получить численное значение некоторого свойства программного обеспечения S или его спецификаций [16], например, количество строк исходного файла, цикломатическая сложность [18], количество ошибок на строку кода, количество классов и интерфейсов, связность и другие.

Тогда метрику изменения программного обеспечения можно определить как разность значений метрики измененного кода MS_{r+1} и метрики исходного кода MS_r:

$$M'\delta_r = MS_{r+1} - MS_r.$$

Зададим набор метрик программного обеспечения $\mathbf{M} = <M_1, M_2, \ldots, M_k>$. Тогда для каждого изменения δ_r можно построить набор метрик изменения

$$\mathbf{M}'\delta_r = <M_1'\delta_r, M_2'\delta_r, \ldots, M_k'\delta_r>.$$

Тогда $\mathbf{M}'\delta_r$ — это точка в k-мерном пространстве кластеризации. Мерой расстояния между точками в этом пространстве выберем евклидово расстояние ρ:

$$\rho(< x_1, x_2, \ldots, x_k >, < y_1, y_2, \ldots, y_k >) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_k - y_k)^2}.$$

Теперь разбиение на кластеры может быть задано следующим образом:

$$C = \{c_1, c_2, \ldots, c_m\}, c_j = \{\delta_r, \delta_g \mid \rho(\mathbf{M}'\delta_r, \mathbf{M}'\delta_g) < \rho_{\min}\},$$

где $\rho_{\min}$ — величина, определяющая меру близости для включения объектов в один кластер.

Алгоритм кластеризации изменений

Пусть известно число кластеров m, выбран набор метрик $\mathbf{M}$ и мера расстояния ρ между точками пространства кластеризации принята евклидовой. В соответствии с методом кластеризации МакКина алгоритм кластеризации изменений следующий.

1. Произвести начальное разбиение множества объектов $\{\delta_r\}$ случайным образом:

$$C^0 = \{c_1, c_2, \ldots, c_n\}, c_i = \{\delta_r \mid \delta_r \notin c_j, j \neq i\}.$$

2. Принять номер итерации $l = 1$.

3. Определить центры кластеров cc_i по формуле

$$cc_i = \frac{\sum_r [\delta_r \in c_i] \overline{M}' \delta_r}{\sum_r [\delta_r \in c_i]}.$$

4. Обновить множества распределения объектов по кластерам $C^l = \{c_i\}$:

$$c_i = \{\delta_r \mid \rho(\mathbf{M}'\delta_r, cc_i) = \min \rho(\mathbf{M}'\delta_r, cc_j)\}.$$

5. Проверить условие $\sum_i \| c_i^l \Delta c_i^{l-1} \| = 0$, где Δ — операция взятия симметрической разности множеств: $A \Delta B = (A \cup B) \setminus (A \cap B)$. Если условия выполнены, то завершить процесс, иначе перейти к шагу 3 с номером итерации $l = l + 1$.

Приведенный алгоритм позволяет автоматизировать процесс разбиения множества изменений на кластеры. В каждый кластер группируются наиболее схожие друг с другом изменения.

Интерпретация результатов кластеризации

Подбор подходящего количества кластеров m и построение функции интерпретации кластеров I_T производится экспертом на основе выборочного анализа изменений, принадлежащих каждому кластеру. Эти задачи значительно менее трудоемки, чем исходная задача, так как на практике имеет смысл различать лишь небольшое число типов изменений.

В процессе построения функции интерпретации кластеров I_T экспертом анализируется изменение исходного кода и комментарий, сопровождающий изменение. В результате устанавливается, какому из типов t_j соответствует данный кластер c_i. При невозможности сопоставления кластера изменений экспертному типу следует повторно обратиться к выбору метрик для кластеризации.

Адекватность классификации

Для оценки согласованности автоматизированной и экспертной классификации в работе [3] используется коэффициент Кохена [4]. Коэффициент Кохена представляет собой меру согласия, с которой два эксперта конкурируют в своих сортировках N элементов по k взаимно исключающим категориям. Эксперта в данном контексте может представлять человек или множество людей, которые коллективно распределяют N элементов, или некоторый алгоритм, который распределяет элементы на основе некоторого критерия.

Выражение для расчета коэффициента согласия Кохена следующее:

$$\kappa = \frac{p_\alpha - p_e}{1 - p_e},$$

где p_α — относительное наблюдаемое согласие между экспертами; p_e — вероятность обусловленности этого согласия случайностью. Если эксперты находятся в абсолютном согласии между собой, тогда $\kappa = 1$. Если же согласие между экспертами отсутствует (но не по причине случайности), тогда $\kappa \le 1$.

По результатам эксперимента [3] получено значение $\kappa = 0{,}79$, которое лежит на границе значительной и превосходной степени согласованности двух методов классификации [5].

Практическое применение метода автоматизированной классификации

Описанный в работе метод может быть использован участниками практически любого проекта по разработке программного продукта. Инструмент, разработанный для применения метода на практике, в момент публикации поддерживает только один тип системы контроля версий — Subverison [19] и языки программирования C++, C#, для которых возможен расчет метрик [16, 17]: цикломатической сложности *CC* [18]; эффективного количества строк кода (без учета пустых строк и комментариев) *eLOC*; общего количества классов и структур *CS*.

Использование автоматизированной классификации изменений программного кода в процессе разработки проекта Navi-Manager позволило сократить время на просмотр исходного кода и повысить его качество, оперативно разрешать запросы на список новой функциональности в конкретных версиях без привлечения разработчиков, а также выявить существующую проблему эффективности разработки.

В результате применения автоматизированной классификации изменений исходного кода для анализа проекта Navi-Manager был достигнут значительный уровень согласованности экспертной и автоматизированной классификации.

При использовании метода проявляется проблема смешанных изменений, сочетающих в себе разнородные модификации кода. Настоящим методом не всегда возможна корректная классификация таких изменений. Нужно заметить, что наличие смешанных изменений на практике нежелательно и даже вредно. При их наличии усложняется процедура просмотра кода и другая работа с историей программного продукта. Выделение смешанных изменений в отдельный тип в процессе кластеризации и решение других проблем метода является целью дальнейших исследований.

Еще одно направление дальнейших исследований — анализ устойчивости метода кластеризации и учет эволюции характера изменений в программном коде при анализе длительных промежутков времени разработки проекта.

Преимущества описанного метода классификации изменений по сравнению с другими методами классификации изменений состоят в следующем:

— *объективность*: для анализа используется исходный код, а не, например, комментарий, сопровождающий изменение. Оценка по исходному коду адекватна в отличие от классификации комментариев к изменениям, ведь комментарии могут не в полной мере соответствовать характеру изменений [20];

— *настраиваемость*: множество метрик программного кода выбирается в зависимости от того, по каким аспектам изменений предполагается группировка;

— *адаптивность*: при кластеризации задается лишь результирующее количество групп. Следовательно, для каждого отдельно взятого проекта предложенный метод позволяет выделить специфичные множества изменений, которые затем интерпретируются как те или иные семантические группы изменений;

— *формальность*: классификация изменений производится с помощью формальных статистических методов.

Заключение

В данной работе было предложено использовать автоматизированную классификацию изменений программного кода в управлении процессом разработки программного продукта. Применение автоматизированной классификации позволяет повысить эффективность и качество процесса обзора исходного кода, а также дает возможность автоматизации контроля изменений, вносимых на ответственных этапах разработки. Предлагается применять автоматизированную классификацию изменений программного кода для предоставления списка новой функциональности в конкретной версии продукта для тестировщика, а также построения отчетов распределения изменений по типам для менеджера проекта.

Литература

1. **Барсегян А. А., Куприянов М. С., Степаненко В. В., Холод И. И.** Технологии анализа данных: Data Mining, Visual Mining, Text Mining, OLAP. СПб.: БХВ-Петербург, 2007.

2. **Мандель И. Д.** Кластерный анализ. М.: Финансы и статистика, 1988.

3. **Князев Е. Г., Шопырин Д. Г.** Автоматизированная классификация изменений программного кода методами многомерного статистического ана-

лиза // Информационные технологии. 2008. № 4. С. 10–15. В печати.

4. **Cohen J.** A Coefficient of Agreement for Nominal Scales // Educational and Psychological Measurement. 1960. P. 37–46.

5. **Emam E. K.** Benchmarking Kappa for Software Process Assessment Reliability Studies // Technical Report ISERN-98-0. International Software Engineering Research Network, 1998. http://citeseer.ist.psu.edu/elemam98benchmarking.html

6. **TortoiseSVN.** A Subversion client, implemented as a windows shell extension. http://tortoisesvn.tigris.org

7. **Navi-Manager Vessel Monitoring System.** http://www.transas.com/products/shorebased/manager/ http://www.transas.ru/products/shorebased/fleet/navi-manager/

8. **KDE.** A powerful Free Software graphical desktop environment for Linux and Unix workstations. http://www.kde.org

9. **Фаулер М.** Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2003.

10. **Kagdi H., Collard M., Maletic J.** Towards a Taxonomy of Approaches for Mining of Source Code Repositories // ACM SIGSOFT Software Engineering Notes: Proc. of the 2005 Intern. Workshop on Mining Software Repositories MSR '05. St. Louis, Missouri, 2005. P. 1–5.

11. **Hassan A. E., Holt R. C.** Source Control Change Messages: How Are They Used And What Do They Mean? 2004. http://www.ece.uvic.ca/~ahmed/home/pubs/CVSSurvey.pdf

12. **Mockus A., Votta L. G.** Identifying reasons for software change using historic databases: Proc. of the Intern. Conf. on Software Maintenance (ICSM). San Jose, California, 2000. P. 120–130.

13. **Demeyer S., Ducasse S., Nierstrasz O.** Finding refactorings via change metrics: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00). 2000. P. 166–178.

14. **Raghavan S., Rohana R., Podgurski A., Augustine V.** Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases: Proc. of 20th IEEE Intern. Conf. on Software Maintenance (ICSM '04). Chicago, Illinois, 2004. P. 188–197.

15. **Maletic J. I., Collard M. L.** Supporting Source Code Difference Analysis: Proc. of IEEE Intern. Conf. on Software Maintenance (ICSM '04). Chicago, Illinois, 2004. P. 210–219.

16. **Орлов С. А.** Технологии разработки программного обеспечения: Учебник для вузов. СПб.: Питер, 2004.

17. **Липаев В. В.** Выбор и оценивание характеристик качества программных средств: Методы и стандарты. М.: Синтег, 2001.

18. **McCabe T. J.** A Complexity Measure // IEEE Trans SE-2. 1976. N 4.

19. **Collins-Sussman B., Fitzpatrick B. W., Pilato M.** Version Control with Subversion. O'Relly. 2004. http://svnbook.red-bean.com/

20. **Zimmermann T., Weigerber P., Diehl S., Zeller A.** Mining Version Histories to Guide Software Changes: Proc. of 26th Intern. Conf. on Software Engineering (ICSE '04). Edinburgh, Scotland, United Kingdom, 2004. P. 563–572.