Министерство образования и науки Российской Федерации
Государственное образовательное учреждение высшего профессионального образования Санкт-Петербургский государственный университет информационных технологий, механики и оптики

На правах рукописи

Князев Евгений Геннадьевич

Автоматизированная классификация изменений исходного кода на основе кластеризации метрик в процессе разработки программного обеспечения

Специальность 05.13.11. Математическое и программное обеспечение вычислительных систем

Диссертация на соискание ученой степени кандидата технических наук

Научный руководитель –
доктор технических наук,
профессор А.А. Шалыто

Санкт-Петербург
2009

Оглавление

Введение 4

Глава 1. Обзор методов классификации изменений исходного кода 10

1.1. Эвристические методы 10

1.1.1. Метод классификации по тексту сопровождающего изменение сообщения 11

1.1.2. Метод поиска и классификации рефакторинга на основе значений определенных метрик 14

1.1.3. Достоинства и недостатки эвристических методов 21

1.2. Синтаксические методы 21

1.2.1. Метод сравнения синтаксических деревьев версий кода 22

1.2.2. Метод анализа синтаксической разницы версий кода с помощью встраиваемых в код тегов 26

1.2.3. Метод, основанный на реализации системы контроля версий, которая хранит абстрактные синтаксические деревья кода, полученные на основе данных из среды разработки 27

1.2.4. Достоинства и недостатки синтаксических методов 32

1.3. Методы, основанные на data mining 32

1.3.1. Метод классификации изменений по признаку возможного наличия в них ошибки 33

Выводы по главе 1 36

Глава 2. Метод автоматизированной классификации изменений на основе кластеризации метрик 38

2.1. Исследование возможности автоматизации классификации изменений исходного кода методом кластеризации метрик 39

2.2. Метод автоматизированной классификации изменений исходного кода 41

2.2.1. Экспертная настройка 42

2.2.2. Вычисление векторов метрик изменений 43

2.2.3. Кластеризация векторов метрик изменений 48

2.2.4. Оценка критериев качества кластеризации 51

2.2.5. Экспертное сопоставление кластеров изменений классам 52

2.2.6. Автоматическая классификация изменений на основе сопоставления кластеров классам 56

2.2.7. Оценка критериев качества метода 56

Выводы по главе 2 61

Глава 3. Применение автоматизированной классификации изменений в процессе разработки программного обеспечения 62

3.1. Использование автоматизированной классификации изменений в проекте разработки программной системы 67

3.1.1. Модель организации программной системы 68

3.1.2. Использование автоматизированной классификации изменений в проекте разработки программной системы 76

3.2. Результаты использования метода 84
3.2.1. Внедрение метода при разработке системы *Navi-Manager* 85
3.2.2. Применение метода при доработке системы *NHibernate* 89
3.2.3. Внедрение метода при разработке компонент системы *LRIT* 96
3.2.4. Внедрение метода при разработке системы *e-Tutor 5000* 99
3.2.5. Использование метода при анализе истории разработки системы *Subversion* 102
3.2.6. Общий эксперимент 107
3.2.7. Выводы по результатам экспериментов 109
3.3. Описание программного инструмента автоматизированной классификации изменений 110
3.4. Особенности реализации используемого алгоритма кластеризации в программном средстве CLUTO 111
Выводы по главе 3 111
Заключение 115
Темы перспективных исследований 117
Список литературы 119

Введение

Актуальность проблемы. Современные организации-разработчики программного обеспечения работают с очень большим объемом исходного кода, что усложняет его понимание и анализ, а, как следствие, затрудняет контроль его качества. В процессе контроля качества программного обеспечения важную роль имеет экспертиза исходного кода (*code review*).

В ходе экспертизы просматривается код с целью обнаружения таких недостатков как, например, алгоритмические и архитектурные ошибки, нарушение принятого стиля кодирования, неясное назначение фрагментов кода. Кроме того, эксперт обычно осуществляет поиск неиспользуемого кода, отслеживает внесение избыточных или несвоевременных изменений, а также внесение изменений, потенциально способных нарушить работоспособность системы, усложнить ее дальнейшее развитие. Экспертиза исходного кода позволяет на ранних стадиях разработки обнаруживать ошибки, которые иначе были бы найдены только на этапе тестирования. Применение экспертизы исходного кода на практике обычно требует существенных временных затрат.

Для упрощения экспертизы кода часто ограничиваются только экспертизой его изменений, так как разработка кода обычно происходит итеративным путем и сводится к внесению изменений (включая новую функциональность). Использование информации о модификациях исходного кода упрощает его понимание за счет концентрации внимания эксперта. Благодаря повсеместному использованию систем контроля версий, при разработке большинства программ доступна история изменений. Однако экспертиза изменений обычно затруднительна из-за их большого числа и ограничения на время работы эксперта. Поэтому приходится проводить выборочную экспертизу изменений. Критерием выбора изменений может быть принадлежность к некоторому классу.

Целесообразно выделять классы изменений, такие как, например, реализация новой функциональности, удаление неиспользуемого кода,

рефакторинг, исправление логики, форматирование кода. Классификация изменений выполняется не только для понимания исходного кода, как отмечено выше, но и для оценки качества изменений по коду, а не с помощью тестирования.

Классификация требуется для контроля процесса разработки, так как, например, если продукт стабилизирован, то никакие изменения, кроме исправления ошибок, проводить не следует. Классификация позволяет также автоматизировать передачу информации между участниками процесса разработки. Она используется для того, чтобы сформировать список изменений, которые требуется проверять, а при необходимости описывать.

В работе предлагается автоматизированный метод классификации изменений исходного кода, состоящий из двух шагов – кластеризации и сопоставления кластеров классам. Распределение изменений по кластерам осуществляется автоматически. Сопоставление их классам выполняет эксперт. Автоматизация распределения изменений по кластерам существенно сокращает время экспертизы изменений кода.

Задача автоматизации классификации изменений исходного кода решалась многими исследователями: *A. Hassan, R. Holt, S. Demeyer, S. Ducasse, S. Raghavan, R. Rohana, J. Maletic, M. Collard, R. Robbes, S. Kim, J. Whitehead* и другими. Ими были разработаны методы классификации изменений на базе *эвристического, синтаксического, метрического* и *Data Mininig* подходов к анализу изменений.

Недостатком *эвристических* методов является их некорректная работа при несоответствии входных данных построенным предположениям. Недостатками *синтаксических* методов являются зависимость от языка программирования и высокая алгоритмическая сложность. Недостатком методов, основанных на *Data Mining,* является сложность создания обучающего множества. Кроме того, большинство перечисленных методов обладает общим недостатком – они предназначены для построения единственной классификации и не допускают настройки на другие классификации.

Указанные недостатки могут быть устранены при совместном использовании метрического подхода и метода кластеризации, который не требует формирования обучающего множества. Поэтому, как следует из изложенного выше, разработка метода автоматизированной классификации изменений с использованием кластеризации и метрического подхода является актуальной задачей.

Применение метода на практике позволяет улучшить качество кода благодаря повышению эффективности процесса его экспертизы. Используя предлагаемый в работе подход, процесс экспертизы в условиях ограничения времени можно строить более эффективно с помощью отбора изменений наиболее важных классов изменений.

Цель диссертационной работы – разработка метода автоматизированной классификации изменений кода в процессе создания программного обеспечения на основе кластеризации метрик.

Основные задачи исследования:

- обоснование возможности частичной автоматизации классификации изменений исходного кода методом кластеризации метрик;
- разработка метода автоматизированной классификации изменений исходного кода на основе кластеризации метрик изменений;
- внедрение результатов работы в практику разработки программного обеспечения.

Научная новизна. На защиту выносятся результаты, обладающие научной новизной.

1. Обоснование возможности частичной автоматизации классификации изменений исходного кода методом кластеризации метрик за счет формулировки гипотезы об автоматизированной классификации и ее экспериментального подтверждения.
2. Обоснование выбора метода *k-средних* с *мерой близости* объектов для кластеризации, *основанной на косинусе угла между векторами метрик* изменений.

3. Метод автоматизированной классификации изменений исходного кода на основе кластеризации метрик изменений, позволяющий сократить число изменений для классификации, выполняемой вручную.

Перечисленные результаты получены в ходе выполнения работ в СПбГУ ИТМО и ЗАО «Транзас Технологии» (Санкт-Петербург).

Методы исследования. В работе использованы методы кластерного анализа, математической статистики и программной инженерии.

Достоверность научных положений, выводов и практических рекомендаций, полученных в диссертации, подтверждается корректным применением методов кластерного анализа и совпадением результатов автоматизированной и экспертной оценки в пределах заданной точности.

Практическое значение работы состоит в том, что все полученные результаты используются в настоящее время и будут использоваться в дальнейшем для повышения качества программного обеспечения в ходе разработки сложных программных комплексов. Предложенный подход применялся для классификации изменений исходного кода в продуктах, разрабатываемых ЗАО «Транзас Технологии» (система мониторинга мобильных объектов *Navi-Manager*, набор компонент глобальной системы *LRIT* (*Long-Range Identification and Tracking*) для отслеживания положения судов в мировом океане, система контроля действий студентов на тренажерах *e-Tutor 5000*), а также в двух программных системах с открытым кодом.

Внедрение результатов. Результаты, полученные в диссертации, внедрены в указанных системах *Navi-Manager, LRIT, e-Tutor 5000*, а также в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО по курсу лекций «Современные технологии разработки программного обеспечения».

Апробация результатов. Основные положения диссертационной работы докладывались на научно-методической конференции «Телематика-2007» (СПб., 2007), X международной конференции по мягким вычислениям и измерениям (СПб., 2007), на конференции «Software Egineering Conference

(Russia) 2007» (М., 2007), XXXVI научной и учебно-методической конференции профессорско-преподавательского и научного состава СПбГУ ИТМО (СПб., 2007), на семинаре Российского Северо-Западного регионального отделения IEEE по компьютерным технологиям и инженерному менеджменту (IEEE Region 8 Russia North-West Computer Society/Engineering Management Society Joint Chapter) (СПб., 2007), IV и V Межвузовской конференции молодых ученых (СПбГУ ИТМО, 2007, 2008), XV Международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке» (СПб., 2008).

Публикации. По теме диссертации опубликовано 11 печатных работ, в том числе две статьи в журналах из списка ВАК. Результаты, приводимые в диссертации, опубликованные без соавторов, получены лично автором. В работах под номерами 1 и 7 в списке публикаций автором предложены способы использования автоматизированной классификации изменений. В работе 6 автором предложен способ расчета метрики покрытия изменения кода модульными тестами. В работах 2, 3 и 8 автором предложен метод автоматизированной классификации изменений на основе предложенного автором способа расчета метрик изменений и их кластеризации. Остальные результаты в статьях под номерами 1, 2, 3, 7 и 8 принадлежат соавтору.

Структура диссертации. Диссертация изложена на 126 страницах и состоит из введения, трех глав и заключения. Список литературы содержит 95 наименований. Работа иллюстрирована 21 рисунком и содержит 34 таблицы.

В первой главе приведен обзор состояния проблемы классификации изменений исходного кода. Сформулированы достоинства и недостатки известных методов, а также задачи, которые должны быть решены в диссертации.

Во второй главе исследована возможность автоматизации классификации изменений исходного кода методом кластеризации метрик. Обоснован выбор метода *k*-средних с мерой близости объектов для кластеризации, основанной на косинусе угла между векторами метрик изменений.

Сформулирована гипотеза: *автоматизированная классификация изменений кода некоторой программной системы возможна методом кластеризации метрик*.

В третьей главе показано, для каких значений критериев качества на практике выполняется приведенная гипотеза. В этой главе приведено подробное описание вариантов применения автоматизированной классификации изменений в процессе разработки программ.

Также в третьей главе описано внедрение в компании *ЗАО* «*Транзас Технологии»* автоматизированной классификации изменений исходного кода на основе кластеризации метрик. Внедрение выполнено (о чем свидетельствуют акты внедрения) при создании:

1. Системы мониторинга мобильных объектов *Navi-Manager;*
2. Компонент системы глобального мониторинга флота *LRIT;*
3. Системы *e-Tutor 5000* контроля действий студентов в процессе обучения на судовом, крановом и других тренажерах.

Предлагаемый метод использовался также при анализе программных систем с открытым кодом: системы контроля версий *Subversion* (*subversion.tigris.org, CollabNET, США*) и объектной обертки над реляционными базами данных *NHibernate* (*JBoss, США*).

В третьей главе описано также разработанное автором программное средство, реализующее предложенный в диссертации метод.

В заключении описаны полученные в диссертации результаты.

Перечисленные результаты получены в ходе выполнения совместных работ *СПбГУ ИТМО* и *ЗАО «Транзас Технологии»* и используются как при разработке программного обеспечения сложных систем, так и в учебном процессе.

Глава 1. Обзор методов классификации изменений исходного кода

Среди существующих методов классификации изменений исходного кода можно выделить следующие группы [59]:

- *неформальные (эвристические) методы* – такие, как метод поиска характерных слов в комментариях к изменениям [55, 75] и метод поиска и классификации рефакторингов на основе значений определенных метрик [48];
- *методы анализа синтаксиса изменений* – такие, как метод сравнения синтаксических деревьев версий кода [83], метод анализа синтаксической разницы версий кода с помощью встраиваемых в код тегов [73] и метод, основанный на реализации системы контроля версий, которая хранит абстрактные синтаксические деревья кода, полученные на основе данных из среды разработки [85];
- *методы, основанные на data mining* [4, 5, 31, 33] – такие, как метод классификации изменений по признаку возможного наличия в них ошибки [64].

Описание методов классификации изменений, объединенных по указанным группам, приводится в разделах 1.1 – 1.3 данной главы.

1.1. Эвристические методы

Эвристические методы основываются на некоторых предположениях при классификации изменений исходного кода. Однако не всегда такие предположения подтверждаются на практике. В диссертации приведено описание следующих эвристических методов классификации изменений исходного кода: метод классификации по тексту сопровождающего изменение

сообщения, метод поиска и классификации рефакторинга на основе значений определенных метрик. Описание данных методов приведено ниже.

1.1.1. Метод классификации по тексту сопровождающего изменение сообщения

В работе [55] предложен метод автоматической классификации изменений кода, основанный на анализе комментариев к ним. Комментарий к каждому изменению задается разработчиком при его внесении в систему контроля версий.

Изменения делятся на следующие классы: *исправление ошибки*, *реализация новой функциональности*, *общая поддержка кода*.

В комментариях к изменениям производится поиск слов, специфичных для каждого из классов изменений. Если вхождение слова найдено, изменению сопоставляется класс, описываемый этим словом. Например, слова *fixed (исправлено), bug (ошибка)* характеризуют класс *исправление ошибки*, а слова *added* (*добавлено), implemented (реализовано)* – класс *реализации новой функциональности*.

В работе [4] анализировались изменения программного кода нескольких программных систем, приведенных в табл. 1.

Таблица 1. Анализируемые методом классификации комментариев к изменениям приложения

Приложение	Тип приложения	Дата начала разработки	Язык программирования
NetBSD	Операционная система	Март 1993 г.	C
FreeBSD	Операционная система	Июнь 1993 г.	C
OpenBSD	Операционная система	Октябрь 1995 г.	C

Продолжение табл. 1.

Postgres	Система управления базами данных	Июль 1996 г.	C
KDE	Оконная система	Апрель 1997 г.	C++
Koffice	Набор офисных программ	Апрель 1998 г.	C++

Из каждого из приведенных проектов случайным образом было выбрано по 18 изменений, причем шесть из них относились к реализации новой функциональности, шесть – к исправлению ошибки и шесть – к общей поддержке кода. Отобранные таким образом 108 изменений были разбиты случайным образом на две группы по 54 изменения. Анализ первой группы нас здесь интересовать не будет.

Для второй группы оценивалась степень согласованности автоматической и экспертной классификации. Предварительно из этих 54 изменений были исключены 11, для которых не совпали классификации двух групп экспертов. Для оставшихся 43 изменений был проведен анализ согласованности автоматической и экспертной классификации, путем измерения коэффициента согласованности по методу *Кохена* [42, 50]. Подробно расчет согласованности классификаций данным методом описан в разд. 2.2.7 второй главы. Значение коэффициента κ оказалось равным *0.71*, что соответствует *значимой степени согласованности* классификаций. При этом совпадение автоматической и экспертной классификации наблюдалось для *81%* от общего числа проанализированных изменений. В табл. 2 приведено распределение 43 анализируемых изменений по автоматическим и экспертным классам.

Таблица 2. Распределение 43 анализируемых изменений по автоматическим и экспертным классам

	Автоматическая классификация			
Экспертная классификация	Общая поддержка кода	Исправление ошибки	Реализация новой функциональности	Всего
Общая поддержка кода	14	1	2	17
Исправление ошибки	2	14	3	16
Реализация новой функциональности	0	0	7	7
Всего	14	15	12	43

Существуют и другие методы анализа текста комментариев к изменениям, в частности [75]. Особенностью данного метода является более строгая обработка текстов комментариев к изменениям, в частности, предварительная нормализация текста, поиск словоформ, частотный анализ используемых терминов. В работе [75] производится классификация изменений по следующим типам: *добавление новой функциональности, устранение дефектов, реструктуризация кода* для упрощения будущих изменений, а также *переработки кода в ходе инспекций*, которые представляют собой смесь корректирующих и улучшающих изменений. Дополнительно производится анализ метрики сложности изменения, устанавливается корреляция значения метрики сложности изменения и его типа.

Недостатком методов классификации изменений на основе текстовых комментариев к ним является то, что он *основан на анализе текста, написанного на естественном языке*. Разработчики не всегда исчерпывающе и

корректно описывают произведенные ими модификации кода. Вообще, содержание комментария субъективно, и текст, написанный человеком, может быть интерпретирован различным образом. Поэтому на практике не всегда корректно судить о содержании изменений, основываясь лишь на комментариях к ним.

1.1.2. Метод поиска и классификации рефакторинга на основе значений определенных метрик

В работе [48] предлагается метод поиска и определения следующих классов рефакторинга:

- *Создание шаблонных методов.* Эта класс рефакторинга заключается в делении методов на меньшие части, чтобы отделить общее поведение от специализированных частей, чтобы дочерние классы могли их перегружать. Необходимость в проведении данного рефакторинга возникает, когда требуется повысить долю повторно используемого кода и когда нужно удалить дублирующуюся функциональность.
- *Выделение композиционных отношений объектов*. Этот класс рефакторинга заключается в перемещении функциональности в созданные для этой цели классы с установкой композиционной связи с ними. Обычно такой рефакторинг производится, чтобы добиться более четкого разделения ответственности классов, а также уменьшить их связность.
- *Оптимизация иерархий наследования классов.* Данный класс рефакторинга производится путем добавления или удаления классов из иерархии наследования и соответствующего перераспределения функциональности. Применяется для упрощения интерфейсов и удаления дублирующей функциональности.

Для поиска и распределения изменений по указанным классам рассчитываются метрики размера метода, метрики размера класса и метрики иерархии. Набор использованных метрик приведен в табл. 3.

Таблица 3. Метрики, использованные в работе [48]

Метрика	Описание
Метрики размера метода	
Mthd-MSG	Число отправленных сообщений (вызванных методов) в теле метода. В работе [71] эта метрика обозначается NOM
Mthd-NOS	Число операторов в теле метода [71]
Mthd-LOC	Число строк кода в теле метода [71]
Метрики размера класса	
NOM	Число методов в классе [41]
NIV	Число переменных экземпляра класса (число полей класса) [71]
NCV	Число переменных класса (число статических полей класса) [71]
Метрики наследования	
HNL (DIT)	Уровень вложенности иерархии [71] (глубина дерева наследования [41])
NOC	Количество непосредственных дочерних классов [41]
NMI	Количество наследуемых методов [71]
NMO	Количество переопределенных методов [71]

Для приведенных метрик используются следующие допущения:

- уменьшение значения метрики размера метода в результате изменения программного кода – признак расщепления данного метода;
- изменение значения метрики размера класса в результате изменения программного кода может быть симптомом переноса функциональности в связанные классы, а также частью симптома

оптимизации иерархии классов, так как этот процесс затрагивает переменные экземпляров классов и их методы;

- изменение значений метрик наследования классов в результате изменения программного кода – это признак оптимизации иерархии классов.

Стоит заметить, что эвристики могут не учитывать существенных параметров исследуемого явления. Поэтому, метод определения рефакторинга на основе эвристик подвержен ошибкам. В частности, приведенные выше допущения не всегда справедливы. Например, уменьшение значения метрики размера метода могло произойти не из-за его расщепления, а из-за удаления из него части функциональности, которая перестала использоваться.

Для обнаружения наличия рефакторинга в изменении применяются следующие эвристические признаки:

- *выделение предка класса;*
- *объединение с классом-предком;*
- *выделение потомка класса;*
- *объединение с потомком класса;*
- *перенос функциональности в другой класс* (предок, потомок или связанный класс);
- *разделение метода;*
- *факторизация общей функциональности.*

В табл. 4 приведено соответствие указанных классов рефакторинга приведенным эвристическим признакам.

Таблица 4. Соответствие классов изменений эвристическим признакам

Класс рефакторинга	Эвристический признак
Оптимизация иерархий наследования классов	Выделение предка класса
	Объединение с классом-предком
	Выделение потомка класса
	Объединение с потомком класса
Выделение композиционных отношений объектов	Перенос функциональности в другой класс
	Разделение метода
Создание шаблонных методов	Факторизация общей функциональности

Рефакторинг *выделение предка класса* изображен на рис. 1. Символами *A, B* изображена иерархия классов до изменения. Символами *A', B', X* изображены классы после выделения предка *X* класса *A*.

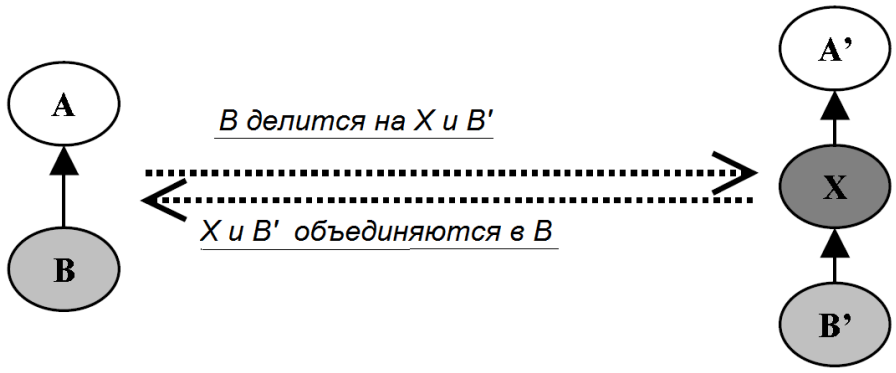

Рис. 1. Изображение выделения предка класса, объединения с классом-предком

Суть эвристического признака данного рефакторинга состоит в следующем.

Если возрос уровень вложенности иерархии классов в результате изменения программного кода *(ΔHNL(B') > 0)* **и** выполнилась одна из следующих альтернатив: уменьшилось количество методов в классе

(ΔNOM(B') < 0) **или** уменьшилось количество переменных-членов класса *(ΔNIV(B') < 0)* **или** уменьшилось количество переменных класса, определяемых классом *B' (ΔNCV(B') < 0)*, **то** имеет смысл говорить о совершенном в процессе изменения программного кода выделении предка класса.

Здесь и далее по тексту использовано следующее сокращение:

$$\Delta M(S') = M(S') - M(S),$$

где M – метрика исходного кода, S – анализируемая синтаксическая единица исходного кода (класс, метод), S' – измененный исходный код, S – исходный код до изменения.

Эвристический признак *PSB(B') выделения предка класса B'* записывается следующим образом:

$$PSB(B') = (\Delta HNL(B') > 0) \wedge ((\Delta NOM(B') < 0) \vee (\Delta NIV(B') < 0) \vee (\Delta NCV(B') < 0)).$$

Рефакторинг *объединение с классом-предком* также проиллюстрирован на рис. 1. Символами *A', X, B'* изображена иерархия классов до изменения. Символами *A, B* изображены классы после объединения предка *X* с классом *A'*.

Суть эвристического признака данного рефакторинга состоит в следующем.

Если уменьшился уровень вложенности иерархии классов в результате изменения программного кода *(ΔHNL(B') < 0)* **и** выполнилась одна из следующих альтернатив: возросло количество методов в классе *(ΔNOM(B') > 0)* **или** возросло количество переменных-членов класса *(ΔNIV(B') > 0)* **или** возросло количество переменных класса, определяемых классом *B' (ΔNCV(B') > 0)*, **тогда** имеет смысл говорить о совершенном в процессе изменения программного кода объединении с классом-предком.

Эвристический признак *MSB(B)* рефакторинга *объединение с классом-предком B*, записывается так:

$$MSB(B) = (\Delta HNL(B) < 0) \wedge ((\Delta NOM(B) > 0) \vee (\Delta NIV(B) > 0) \vee (\Delta NCV(B) > 0)).$$

Следующие два вида рефакторинга – *выделение потомка класса A' и объединение с потомком класса* проиллюстрированы на рис. 2.

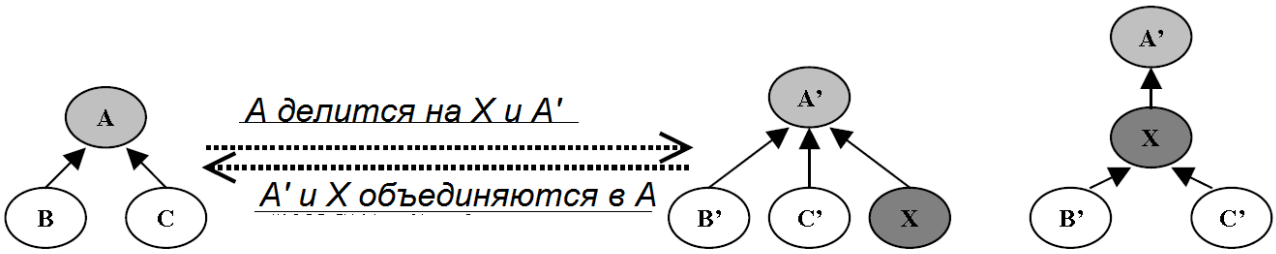

Рис. 2. Изображение выделения потомка класса и объединения с потомком класса

В первом случае из класс *A* разделяется на классы *A'* и *X*, а затем в класс *X* выделяется вся общая функциональность из классов *B'* и *C'*. Во втором случае, наоборот, в классы *B', C'* перемещается функциональность из класса *X*, который затем удаляется.

Эвристический признак *SSU(A')* для рефакторинга *выделение потомка класса A'* и эвристический признак *объединение с потомком класса MSU(A)* приведены ниже:

$$SSU(A')=(\Delta NOC(A')\neq 0)\wedge((\Delta NOM(A')<0)\vee(\Delta NIV(A')<0)\vee(\Delta NCV(A')<0));$$

$$MSU(A)=(\Delta NOC(A)\neq 0)\wedge((\Delta NOM(A)>0)\vee(\Delta NIV(A)>0)\vee(\Delta NCV(A)>0)).$$

Рефакторинг *перенос функциональности в другой класс (предок, потомок или связанный класс) MTOC(B')* изображен на рис. 3.

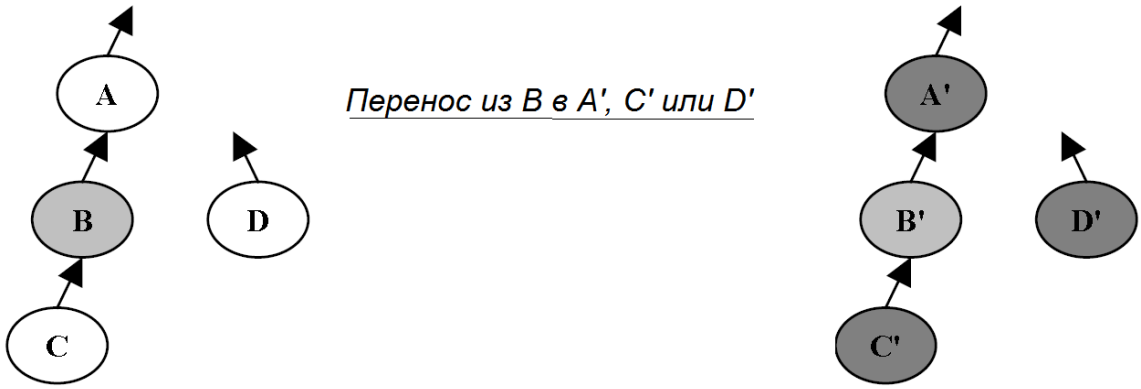

Рис. 3. Изображение переноса функциональности в другой класс (предок, потомок или связанный класс)

Символами *A, B, C* обозначены классы, состоящие в отношении наследования, символом *D* – класс, связанный с классом *B* до проведения рефакторинга. В процессе выполнения рефакторинга часть функциональности

из класса *B* переносится в класс *A*, *C* или *D*, образуя соответственно классы *B'*, *A'*, *C'*, *D'*.

Данный рефакторинг определяется следующим эвристическим признаком:

$$MTOC(B')=((\Delta NOM(B')<0)\vee(\Delta NIV(B')<0)\vee(\Delta NCV(B')<0))\wedge$$

$$(\Delta NHL(B')=0)\wedge(\Delta NOC(B')=0).$$

Рефакторинг *разделение метода a* изображен на рис. 4. Метод *a* класса *A* до проведения рефакторинга обозначен как *a*. Соответствующий метод после выполнения рефакторинга обозначен как *a'*. В ходе данного рефакторинга часть кода метода *a* выделяется в новый метод *x*.

Рефакторинг *факторизация общей функциональности методов a, b класса A* также изображен на рис. 4. В ходе данного рефакторинга общий код методов *a*, *b* выделяется в новый метод *x*.

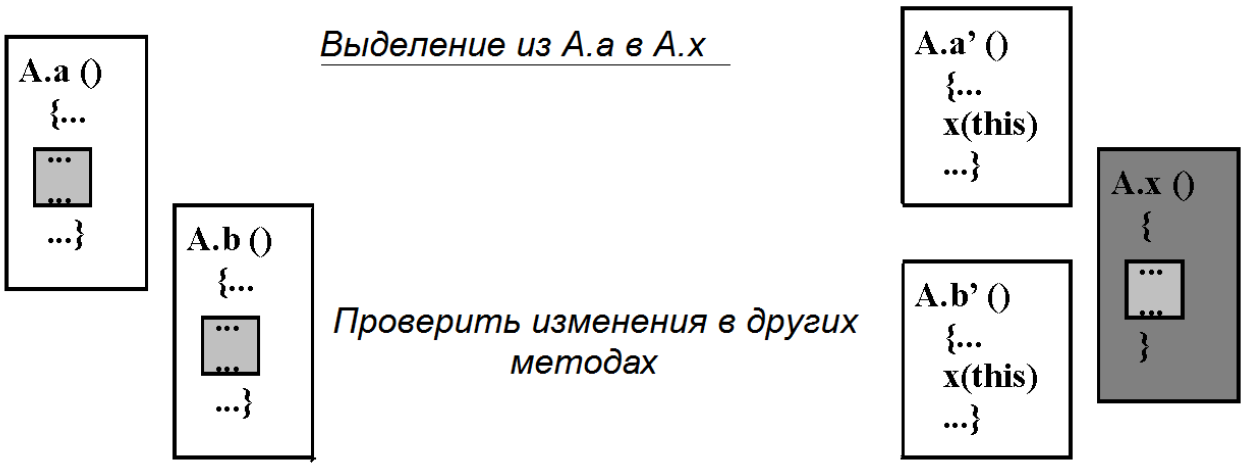

Рис. 4. Изображение разделения метода и факторизации общей функциональности

Эвристический признак *разделение метода a MS(a, T)* записывается в виде:

$$MS(a, T)=(\Delta Mthd\_MSG(a)<T),$$

где *T* – ограничивающий параметр количества переносимых вызовов. Он необходим для выделения рефакторингов, в ходе которых выделено не более *T* вызовов методов.

Эвристический признак *факторизация общей функциональности методов a, b FF(a, T)* записывается в виде:

$$FF(a, b, T) = (\Delta Mthd\_MSG(a) < T) \wedge (\Delta Mthd\_MSG(b) < T),$$

где T – ограничивающий параметр количества переносимых вызовов.

Для классификации изменений по данным работы [48] при поиске заданного типа рефакторинга доля ошибок классификации второго рода (ложное срабатывание эвристики на изменении, которое не является искомым рефакторингом) в общем числе тестовых изменений составляет в среднем от 43% до 91% для различных типов рефакторингов. Такое низкое качество результатов свидетельствует о необходимости улучшения приведенных эвристик при классификации указанных типов рефакторингов.

Описанный метод показывает возможность практического использования метрик для определения различных типов изменений исходного кода, в том числе высокоуровневых операций по изменению архитектуры программной системы.

1.1.3. Достоинства и недостатки эвристических методов

Достоинством эвристических методов является простота их реализации. Это существенно, так как часто полную реализацию решения построить очень сложно, если вообще возможно.

Недостатком эвристических методов является работа с ошибками для некоторых входных данных, так как при разработке эвристик сложно учесть все возможные значения таких данных. Для приведенных методов это выражается в невысокой средней точности определения класса изменения.

1.2. Синтаксические методы

Синтаксические методы классификации изменений основаны на синтаксическом анализе предшествующего изменению и результирующего исходного кода. В диссертации приведено описание следующих

синтаксических методов классификации изменений исходного кода: метод сравнения синтаксических деревьев версий кода, метод анализа синтаксической разницы версий кода с помощью встраиваемых в код тегов, метод, основанный на реализации системы контроля версий, которая хранит абстрактные синтаксические деревья кода, полученные на основе данных из среды разработки. Описание данных методов приведено ниже.

1.2.1. Метод сравнения синтаксических деревьев версий кода

В работе [83] предлагается метод анализа изменений программного кода на основе сравнения абстрактных семантических графов, построенных для оригинальной и измененной версии программных файлов. В процессе анализа производится построение последовательности операций по добавлению, удалению и переносу вершин дерева оригинальной версии таким образом, чтобы получить граф для результирующей версии программного кода. Затем на основе полученной последовательности операций изменение характеризуется по 398 различным пересекающимся признакам. Примеры таких признаков приведены в табл. 5.

Абстрактный семантический граф – это абстрактное синтаксическое дерево с дополнительными узлами, несущими некую семантическую информацию – информацию о типе. Эти узлы соединяют литералы и определения (декларации) с их типами, а ссылки на переменные – на определения их переменных [83].

Для определения разницы между двумя абстрактными семантическими графами выполняется эвристический алгоритм сравнения двух упорядоченных корневых деревьев – абстрактных семантических деревьев, выделенных из исходных абстрактных семантических графов. Абстрактное семантическое дерево получается из абстрактного семантического графа путем удаления вложенных узлов с семантической информацией, добавлением атрибутов узлов и проведением неких дополнительных действий.

Алгоритм функционирует путем итеративного сравнения частей деревьев и пересчета стоимостей соответствий, основываясь на всем абстрактном семантическом графе, пока для всех вершин не будут установлены соответствия. Результатом работы алгоритма для абстрактных семантических деревьев T_1 и T_2 является последовательность операций редактирования дерева T_1, приводящая его к дереву T_2. Последовательность операций редактирования состоит из соответствия узлов, вставки узла и удаления узла. Каждая операция имеет назначенную стоимость. Операции соответствия бывают нескольких типов: *обновление*, *простое соответствие*, *перенос*. Алгоритм учитывает типы узлов – соответствовать друг другу могут только узлы одного типа.

Суть алгоритма состоит в следующем. На каждой итерации он пытается найти все возможные соответствия узлов от корней деревьев T_1 и T_2 вниз к листьям, а затем – вверх. Узлы в T_1, которые не соответствуют ни одному из узлов T_2, считаются удаленными; узлы T_2, которые не соответствуют ни одному из узлов T_1, считаются добавленными. Между шагами работы алгоритма пересчитывается матрица стоимостей. Матрица стоимостей используется на этапе поиска соответствий от листьев к корню и предназначена для поиска способа минимальной трансформации дерева T_1 в T_2 с помощью алгоритмов динамического программирования.

Алгоритм генерации характеристик на основе множества признаков изменений состоит в проверке измененных узлов графов на соответствие заданным условиям и накоплении статистики по каждому признаку для всех проанализированных изменений. Возможно отслеживание как простых признаков, например, изменилось ли тело существующей функции, какие новые функции добавились, так и сложных, например, какие условные операторы были изменены так, что стали использовать переменную, которая уже была в области видимости.

Данный алгоритм реализован в специализированном инструменте *Difference extractor (Dex)* [83] анализа изменений программ на языке *C*. В работе [83] описано применение инструмента для анализа 112 изменений по

исправлению ошибок в HTTP-сервере с открытым исходным кодом *Apache* [92] для *Unix* и *Windows NT* и 71 изменений по исправлению ошибок в наборе компиляторов *Gnu Compiler Collection* (*GCC)* [54] для *Unix*. В табл. 5 приведены результаты анализа данных для шести наиболее часто встречающихся характеристик изменений.

Таблица 5. Частота шести наиболее часто встречающихся характеристик среди изменений по исправлению ошибок для *Apache* и *GCC* в процентах от общего числа проанализированных изменений

Характеристика изменения	*Apache*, %	*GCC*, %
Правка тел существующих функций	94.64	90.14
Вставка условных выражений в тела существующих функций	37.50	43.66
Вставка вызовов функций	37.50	56.34
Правка существующих вызовов функций	33.04	26.76
Правка условий существующих условных выражений	31.25	32.39
Правка существующих выражений присваивания	25.89	36.62

Для данного алгоритма был проведен анализ корректности классификации изменений по всем 398 признакам путем сравнения результатов работы алгоритма и экспертной характеристики набора из 39 случайно отобранных из числа ранее анализировавшихся 112 изменений для *Apache* и 34 случайно отобранных из числа ранее анализировавшихся 71 изменений для *GCC.* Результаты такого сравнения приведены в табл. 6

Таблица 6. Соотношение корректно и некорректно (хотя бы по одному из 398 признаков) охарактеризованных изменений и их характеристик

	Число проанализированных изменений	Число некорректно оцененных изменений	Число некорректно построенных характеристик
Apache	39	3	3
GCC	34	6	21
Всего	73	9	24
Всего, %	100	12	0.08

Итак, в результате анализа корректности оценивания изменений описанным методом выяснилось, что лишь 12% от всех проанализированных изменений содержали одну или более ошибочных характеристик из 398, рассчитывающихся для каждого изменения. Это очень хороший результат. Действительно, отношение ошибочных характеристик к числу построенных в процессе анализа характеристик, составляет 0.08%. Исчерпывающий набор характеристик изменений, а также высокая точность анализа – это преимущества данного метода перед другими методами, приведенными в настоящем обзоре.

К недостаткам метода можно отнести высокую алгоритмическую сложность реализации сравнения синтаксических деревьев версий. Высокая сложность алгоритма – это источник ошибок реализации и часто продолжительное время выполнения программы. Так, указанный алгоритм имеет сложность $O(n^4)$ и требует затрат памяти $O(n^2)$, где n – число вершин в абстрактном синтаксическом дереве исходного кода. Поэтому среднее время выполнения сравнения двух версий файлов для *GCC* составляло *5 мин.*, а для *Apache – 60 с*. Характеристики вычислительной системы, на которой производились тесты, следующие: операционная система – *Windows 2000 Server*; процессор – *Pentium IV Xeon* с частотой *1.8 ГГц*; память – *1 Гб*. При этом анализ истории программной системы, сравнимой по размерам с *GCC*, за месяц

разработки будет требовать порядка 25 ч. вычислений при частоте изменений порядка 10 в день.

1.2.2. Метод анализа синтаксической разницы версий кода с помощью встраиваемых в код тегов

В работах [73, 44] анализируется разница версий с помощью XML-тегов структуры программы, встраиваемых в код перед выполнением сравнения его версий. Благодаря тегам в ходе анализа изменений обнаруживаются добавленные, измененные и удаленные синтаксические вхождения и вычисляются классы изменений. В результате работы метода даются ответы на вопросы: добавились ли новые методы в определенный класс, есть ли изменения в директивах препроцессора, было ли модифицировано условие любого условного оператора и т.д.

В работе [44] приводятся результаты анализа программного средства с открытым кодом для анализа данных *HyppoDraw* [56]. Изменения делились на следующие классы: *изменение комментария, добавление метода, изменение директивы препроцессора, другое изменение*. В указанной работе не приводится сведений относительно качества распределения изменений по указанным классам.

Следует заметить, что описанный метод разрабатывался не столько для классификации изменений, сколько для решения таких задач как адресация конкретных синтаксических и семантических элементов кода, запрашивание информации относительно кода, а также его изменений, а также трансформации кода. Эти возможности позволяют авторам считать данный метод универсальной платформой для построения на ее базе решений по анализу изменений кода более высокого уровня.

Описанный метод включен в обзор данной диссертации с целью полного иллюстрирования известных подходов, которые могут применяться для классификации изменений кода. Учитывая недостаточную проработанность задачи классификации изменений и слишком большую общность метода, в

дальнейшем исключим его из сравнения с предложенным в диссертации методом.

1.2.3. Метод, основанный на реализации системы контроля версий, которая хранит абстрактные синтаксические деревья кода, полученные на основе данных из среды разработки

Популярные среди разработчиков программного обеспечения системы контроля версий *Concurrent Version System (CVS)* [46], *Subversion* [91] имеют два недостатка, ограничивающие информацию об истории модификаций, которое может быть из них извлечено. Это – ориентация на файлы и представление истории модификаций в виде последовательности мгновенных «снимков» – версий.

Отсутствие привязки к предметной области (коду программ) вынуждает применять синтаксический анализ для каждой версии. Таким образом, процесс анализа истории модификаций программного кода становится сложным, ресурсоемким, требует применения специальных инструментов (синтаксических анализаторов).

Системы контроля версий, ориентированные на файлы, хранят историю их изменений, представляя изменения в терминах добавленных и удаленных строк. В этих системах теряется информация о том, какие символы поменялись в строке кода, так как изменение строки в них представляется удалением исходной и добавлением измененной строки [40].

Системы контроля версий хранят информацию об изменениях в виде разницы между предыдущей и следующей версией. Новая версия добавляется по инициативе пользователя, когда он вносит свои изменения в систему. Пользователи систем контроля версий часто выкладывают изменения в систему только после того, как сделан относительно законченный этап работы или в конце дня. В этой ситуации происходит потеря промежуточных шагов в процессе изменения кода. Проблема подхода с внесением информации в систему контроля версий по инициативе пользователя заключается в том, что

вследствие этого может образовываться большая разница между последовательными версиями, а это ведет к сложности различения отдельных изменений.

В работе [85] предлагается представлять историю изменений программной системы в виде *эволюционирующего в процессе изменений абстрактного синтаксического дерева программы*. Такой подход к хранению истории модификаций программ свободен от вышеописанных недостатков систем контроля версий файлов.

В основанном на изменениях, а не версиях, хранилище история программной системы содержится в виде последовательности *операций по изменению*, обязательно приводящих программную систему в актуальное состояние. Эти операции не могут быть выведены достаточно точно путем сравнения двух версий системы [63]. Вместо этого они восстанавливаются путем отслеживания действий пользователя с помощью среды разработки. В систему контроля версий автоматически попадают изменения, сделанные в среде разработки в процессе создания программной системы. Таким образом, интеграция среды разработки с хранилищем истории программной системы позволяет получить максимально полную информацию о модификациях программного кода.

Абстрактное синтаксическое дерево (*АСД*) [63, 3, 27] в хранилище представляет собой эволюционирующую объектно-ориентированную программу. В *АСД* на разных уровнях присутствуют узлы модулей, классов, методов, переменных и выражений. Пример абстрактного синтаксического дерева приведен на рис. 5.

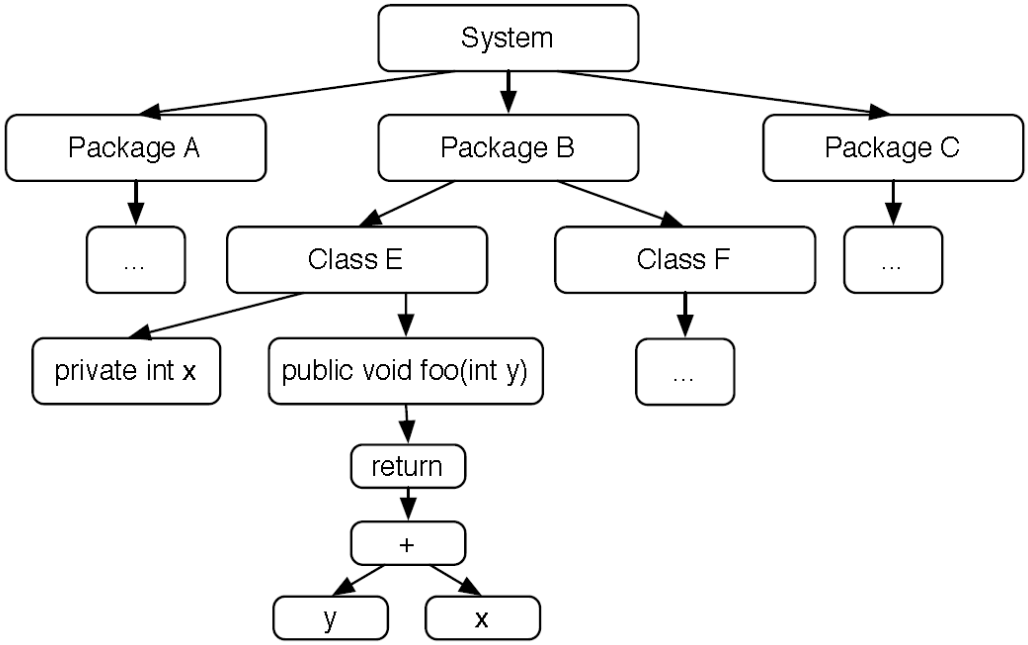

Рис. 5. Пример абстрактного синтаксического дерева программной системы *System* и ее деления на модули (*Package A, B, C*), классы (*Class E, F*), функции (*foo*) и операторы

АСД представляет состояние программы, через которое она проходит в процессе эволюции. Каждый узел в дереве содержит историю всех операций по его изменению. В методе поддерживаются атомарные и составные операции по изменению программной системы.

Атомарные операции:

- создание узла в дереве;
- добавление к заданному узлу существующего узла;
- удаление узла;
- изменение свойств узла.

Составные операции комбинируют набор операций по изменению (атомарных или в свою очередь составных) и вводятся как дополнительный уровень абстракции изменений, необходимый для упрощения анализа системы. Выделяются следующие составные операции по изменениям (классы в терминах данной работы):

- *действия уровня разработчика* – например, добавление метода, изменение описания класса и другие;

- *рефакторинг* – сохраняющее поведение кода изменение, направленное на улучшение его дизайна. Например, переименование метода, выделение метода. Информация о рефакторинге может автоматически отслеживаться средой разработки и попадать в хранилище изменений, если производится с помощью встроенных средств среды разработки (не вручную);
- *исправление ошибки* – набор изменений, ведущих к исправлению определенной ошибки;
- *сессия разработки* – набор изменений, выполняемых в рамках одной логической серии;
- *реализация новой функциональности* – набор изменений, необходимых для реализации определенной функциональности.

Подход с поддержкой полной истории изменений в терминах абстрактного синтаксического дерева применялся для поиска рефакторинга в проектах *SpyWare* [85] (реализация описываемой системы) и *Project X* (некий веб-проект). В табл. 7 приведен перечень рефакторингов, поиск которых проводился среди различных изменений исходного кода, накопленных за период разработки равный нескольким месяцам.

Таблица 7. Обнаруживаемые классы рефакторинга методом, основанным на реализации системы контроля версий

Классы рефакторинга	Ссылка на описание
Добавление параметра в метод	Все анализируемые классы рефакторинга описаны в работе [32]
Вытеснение метода вверх по иерархии	
Переименование класса	
Переименование метода	
Абстрактная инстанция переменной	
Вытеснение переменной экземпляра класса вверх по иерархии	
Выделение метода	

Продолжение табл. 7.

Выделение метода в компонент	Все анализируемые классы рефакторинга описаны в работе [32]
Встраивание метода	
Выделение выражения в переменную	
Встраивание временной переменной	
Переименование переменной экземпляра класса	
Переименование временной переменной	
Превращение временной переменной в переменную экземпляра класса	

Итак, предложенный в работе [85] метод позволяет сохранить полную информацию об истории изменений программной системы, позволяет упростить анализ истории изменений за счет устранения этапа синтаксического разбора программного кода для каждой версии. В работе [85] приводятся примеры поиска различных классов рефакторинга, совмещенных с другими модификациями в пределах одной сессии разработки. В работе [85] решается эта задача, но только в случае использования для проведения рефакторинга встроенных в среду разработки средств.

Недостатком описанного метода является отсутствие возможности поиска и определения класса рефакторинга без поддержки среды разработки. Также отсутствует сравнительный анализ данного метода и других методов анализа рефакторинга вследствие отсутствия детальной информации об изменениях для проектов, которые хранятся с использованием стандартного подхода. В работе [85] не описан способ разрешения конфликтов изменений, которые неизбежно возникнут при использовании хранилища программного кода, основанного на изменениях, в процессе многопользовательской разработке.

Еще одним недостатком метода является необходимость избыточного хранения и обработки полного набора модификаций в системе контроля версий, проводимых пользователем в среде разработки, так как далеко не все они сохраняются в итоговом коде.

Инструменты поддержки описанного метода требуют значительно больших затрат вычислительных ресурсов, чем существующие системы контроля версий файлов, а также дополнительной поддержки перевода используемого языка программирования в абстрактное синтаксическое дерево, и трансляции команд изменения исходных файлов со стороны среды разработки. Поэтому рассчитывать на широкое внедрение данного подхода в настоящее время пока не приходится.

Общие достоинства и недостатки, характерные для всех синтаксических методов классификации изменений, приводятся в следующем разделе.

1.2.4. Достоинства и недостатки синтаксических методов

Достоинством синтаксических методов является их точность и полнота получаемой информации о модификациях. Однако на практике это часто оборачивается недостатком в виде большого количества информации для анализа.

Недостатками синтаксических методов является то, что они в большинстве своем сложны и зависят от конкретного языка программирования, а также не обладают адаптивностью. Применение таких методов оправданно только для анализа простых изменений. В случае если структура изменения сложна, что часто встречается на практике, метод не позволит дать однозначный ответ на вопрос о его классе.

1.3. Методы, основанные на data mining

Data Mining или *добыча данных* [4, 33, 5] – это исследование и обнаружение «машиной» (алгоритмами, средствами искусственного интеллекта) в «сырых» данных скрытых знаний, которые ранее не были известны, нетривиальны, практически полезны, доступны для интерпретации человеком.

Задача классификации изменений сводится к определению класса изменения по его характеристикам. В этой задаче множество классов, к которым может быть отнесено изменение, известно заранее.

1.3.1. Метод классификации изменений по признаку возможного наличия в них ошибки

В работе [64] предлагается метод классификации изменений исходного кода на два класса: изменения, содержащие ошибки и изменения без ошибок. Данный метод основан на кластеризации информации из нескольких источников: исходного кода, метрик сложности кода, метаданных изменения.

Классификация производится методом *SVM (Support Vector Machine)* [58], который хорошо себя зарекомендовал в задаче классификации текстов. Метод *SVM* классифицирует изменения, содержащие ошибку с точностью *P* в среднем *78%* и полнотой *R* в среднем *65%* для 12 проектов.

Величины точности и полноты классификации изменений, содержащих ошибку, связаны с ошибками классификации следующим образом:

$$P = \frac{tp}{tp + fp} \ , \qquad R = \frac{tp}{tp + fn} \ ,$$

где *tp* – («true positive») – число верно классифицированных изменений, *fp* – («false positive») – число ошибок классификации второго рода, *fn* – («false negative») – число ошибок классификации первого рода.

Точность *P* измеряет вероятность того, что случайно выбранное изменение из построенного классификатором множества изменений, принадлежащих классу *c*, действительно принадлежит классу *c*. Полнота *R* измеряет вероятность того, что классификатор присвоит класс *c* случайно выбранному изменению из множества изменений, принадлежащих классу *c*.

Список проанализированных программных систем приведен в табл. 8.

Таблица 8. Список проанализированных программных систем

Программная система	Анализируемые ревизии системы контроля версий	Число изменений без ошибок	Число изменений с ошибками	Изменений с ошибками, %	Число характеристик (features)
Apache HTTP 1.3	500-1000	579	121	17.3	11445
Bugzilla	500-1000	149	417	73.7	10148
Columba	500-1000	1270	530	29.4	17411
Gaim	500-1000	742	451	37.8	9281
GForge	500-1000	339	334	49.6	8996
Jedit	500-750	626	377	37.5	13879
Mozilla	500-1000	395	169	29.9	13648
Eclipse	500-750	592	67	10.1	16192
Plone	500-1000	457	112	19.6	6127
PostgreSQL	500-1000	853	273	24.2	23247
Scarab	500-1000	358	366	50.5	5710
Subversion	500-1000	1925	288	13.0	14856
Всего		8285	3505	29.7	150940

Работа алгоритма состоит из следующих этапов:

- *Поиск изменений, исправляющих ошибки* производился путем обнаружения слов *fixed* (*исправлено*) или *bug* (*ошибка*) или ссылки на запись в системе учета ошибок в комментариях к изменениям.
- Для всех изменений, исправляющих ошибки, производится *поиск изменений, в которых эти ошибки были сделаны* при помощи алгоритма *SZZ* (*Sliwerski, Zimmermann, and Zeller*) [88, 94, 95].
- Для всех изменений *выделение информации из системы контроля версий*. Выделяется несколько характеристик (*features*), включая: длину комментария к изменению, метрику *LOC (lines of code, число*

строк кода) изменения (число добавленных строк кода плюс число удаленных строк кода), метрику *LOC* новой версии программного кода, разницу метрик сложности (*цикломатической сложности*, *максимального уровня вложенности* [29, 23] и других) до изменения и после него. Дополнительно выделяется текстовая информация из комментария к изменению, и изменившегося программного кода с помощью алгоритма *BOW* (*bag-of-words*) [86, 87].

- Набор выделенных характеристик для изменений используется для обучения алгоритма классификации *SVM*. В дальнейшем для каждого нового изменения аналогичным образом рассчитывается описанный набор характеристик и производится классификация данного изменения. В результате классификации изменение относится к классу изменений, содержащих или не содержащих ошибку.

В описываемом исследовании использовалась система *Kenyon* [39] для выделения информации из систем контроля версий *CVS* [46] или *Subversion* [91], в которых содержится программный код исследуемых программных систем.

Недостатки, обнаруженные в работе [64], приведены ниже:

- *все исследуемые системы – это системы с открытым исходным кодом*, что может сказаться на применимости метода классификации в процессе разработки коммерческого программного обеспечения вследствие неучтенности специфичных факторов;
- *данные об исправлении ошибок неполны,* так как не все комментарии к изменениям содержат указания на исправление ошибки в распознаваемом формате, и это ведет к ошибкам классификации;

- для работы метода *требуется накопленная история изменений* (минимум *100* изменений для проанализированных проектов);
- *в системах учета запросов* на изменение, кроме запросов на исправление ошибок, *могут содержаться запросы на реализацию новой функциональности*, что также приведет к ошибкам классификации изменений.

Кроме того, в работе не исследован вопрос устойчивости и адаптивности классификатора, что отражается в необходимости регулярно переобучать метод в процессе использования.

Тем не менее, результат предсказания описанным методом является одним из лучших в настоящее время. При этом метод имеет дополнительное преимущество: он локализует конкретное место в коде (строку кода), в котором предсказывается ошибка. Тогда как большинство методов предсказания ошибок ограничиваются лишь указанием файла или функции.

Недостатком описанного метода, основанного на подходе *data mining*, является необходимость построения обучающего множества, содержащего изменения, порождающие ошибки и их исправляющие. Построение такого множества – нетривиальная задача.

Выводы по главе 1

1. Недостатком рассмотренных в данной главе эвристических методов, является работа с ошибками для некоторых входных данных, так как при разработке эвристик сложно учесть все возможные значения таких данных.
2. Недостатками синтаксических методов является их алгоритмическая сложность, зависимость от языка программирования, а также отсутствие адаптивности. Применение синтаксических методов классификации изменений оправдано только для анализа простых изменений.

Недостатком метода, описанного в разд. 1.2.3, является необходимость избыточного хранения и обработки полного набора модификаций в системе контроля версий, проводимых пользователем в среде разработки, так как далеко не все они сохраняются в итоговом коде.

3. Недостатком метода, основанного на подходе *data mining*, является необходимость построения обучающего множества, содержащего изменения, порождающие ошибки и их исправляющие. Построение такого множества в общем случае с достаточной точностью – нетривиальная задача.
4. Кроме того, указанные эвристические и синтаксические методы обладают общим недостатком – специализированностью. Это проявляется, в том, что в общем случае без существенной модификации метода нельзя перенастроить метод на экспертную классификацию изменений, отличную от исходной.

Из изложенного выше следует, что существующие методы классификации изменений программного кода обладают недостатками, указанными выше. Настоящая работа направлена на их устранение.

Глава 2. Метод автоматизированной классификации изменений на основе кластеризации метрик

В настоящей работе предлагается метод классификации изменений исходного кода на основе кластеризации метрик [13, 15, 16, 17], свободный от указанных выше недостатков, который, в отличие от приведенных методов, можно применять для достаточно широкого круга задач [10, 13, 14, 16, 18, 65]. Например, как описано в разд. 3.1 и разд. «Темы перспективных исследований».

По сравнению с описанными эвристическими методами классификации изменений исходного кода, предложенный метод обладает достоинством – является настраиваемым (набор метрик исходного кода выбирается в зависимости от того, по каким аспектам изменений строится классификация).

Так, например, при добавлении еще одного класса рефакторинга в метод поиска и классификации рефакторинга на основе значений определенных метрик, описанного в разд. 1.1.2, может оказаться сложно, если вообще возможно, подобрать хорошую эвристику.

В отличие от описанных в первой главе методов классификации изменений, предложенный метод не ограничен заданным набором классов. Результат классификации зависит от заданного набора метрик для кластеризации, отражающих интересующие эксперта аспекты классификации, и экспертного сопоставления кластеров классам.

Еще одним преимуществом метода, выражающимся в повышении качества классификации, является способ расчета метрик изменений, основанный не на разности исходной и результирующей метрик, а на основе добавленных, удаленных и измененных строк. На основе экспериментов, проводимых в рамках данной работе, было выяснено, что такой подход работает лучше.

Сравнение эффективности разработанного в диссертации метода с другими методами не выполнялось в связи с различием используемых в них

классификаций, что делает прямое сравнение результатов классификаций некорректным, а настройка на другие классификации для большинства методов невозможна.

В диссертации приводятся результаты сравнения классификаций изменений в программной системе с открытым исходным кодом, выполненные с использованием предложенного автоматизированного метода и вручную. К участию к эксперименту привлекались независимые эксперты, имеющие опыт разработки сложных программных систем не менее пяти лет. Всего в диссертации проанализировано пять программных систем.

В настоящей главе исследована возможность автоматизации классификации изменений исходного кода методом кластеризации метрик. Обоснован выбор метода *k-средних* [5, 25] с мерой близости объектов для кластеризации, основанной на *косинусе угла между векторами метрик изменений*. Предложенный метод позволяет улучшать качество классификации за счет возможности увеличения числа используемых метрик. В настоящее время в методе используется 11 метрик, перечень которых приведен в разд. 2.2. Автор предполагает, что возможно совершенствование качества классификации за счет увеличения числа используемых метрик, а также настройка на другие перечни классов изменений, кроме тех, которые рассмотрены в диссертации.

2.1. Исследование возможности автоматизации классификации изменений исходного кода методом кластеризации метрик

Рассмотрим некую *программную систему P* в процессе ее развития во времени. Состояние этой системы в каждый момент времени t задается ее текущим кодом S_t. Для удобства обозначим множества неизменных состояний S_t, в течение последовательных интервалов времени $t \in (t_{r-1}, \ldots, t_r)$, через S_r, где r – целое число, $1 \leq r \leq N$, N – общее число различных состояний кода.

Изменением исходного кода назовем отображение δ_r, переводящее код из предшествующего состояния S_{r-1} в модифицированное состояние S_r:

$$S_{r-1} \xrightarrow{\delta_r} S_r.$$

С помощью каждого изменения кода разработчиком достигается определенная цель по развитию программной системы. При реализации программных систем на практике становится ясно, что часто цели, достигаемые с помощью различных изменений, имеют много общего между собой [32, 24, 15]. Для упрощения задач экспертизы кода целесообразно делить изменения на классы в соответствии с выбранными целями.

Каждое изменение δ в соответствии с целью его внесения может быть отнесено к некоторому классу c, где $c \in C$. При этом C представляет собой множество классов изменений, специфичное для разрабатываемой программной системы. Наиболее распространенными являются следующие классы изменений: реализация новой функциональности, исправление логики, рефакторинг, удаление избыточного кода, форматирование кода [32, 24, 15].

В общем виде способ классификации множества изменений с трудом поддается формализации. Поэтому задача отнесения изменения к тому или иному классу c трудоемка и требует участия эксперта высокой квалификации. Эксперт определяет состав множества классов, специфичный для каждой программной системы, а также решает, к какому из классов относится конкретное изменение.

Следует заметить, что результат экспертной классификации зависит не только от анализируемой программной системы и набора изменений, но и от конкретного эксперта. Поэтому можно ожидать, что при классификации некоторых изменений разными экспертами возможно различное распределение изменений по заданным классам.

Определим здесь проверочное множество Δ_e как некоторое множество изменений, классифицированное экспертом. Для устранения возможных противоречий в данной работе построение проверочных множеств изменений

производится двумя экспертами, и изменения, классифицированные ими различным образом, из проверочного множества исключаются.

Метод кластеризации метрик изменений позволяет нивелировать субъективность классификации, и, как показано в диссертации, может быть обеспечена частичная автоматизация классификации изменений исходного кода. Автоматизация классификации выполняется на основе следующей гипотезы.

Гипотеза. *Автоматизированная классификация изменений кода некоторой программной системы возможна методом кластеризации метрик*.

Более строго эта гипотеза может быть сформулирована следующим образом: «Для заданного множества классов изменений и требуется классифицировать множество изменений программной системы. Пусть также для каждого изменения построен вектор метрик. Тогда данные векторы метрик можно кластеризовать таким образом, что каждому полученному кластеру будет соответствовать один класс изменений».

Обоснование справедливости данной гипотезы будет приведено в последней главе диссертации на основе экспериментальной ее проверки.

2.2. Метод автоматизированной классификации изменений исходного кода

Автоматизацию классификации изменений исходного кода в данной работе предлагается строить на основе кластеризации метрик изменений.

Схема метода приведена на рис. 6. В качестве входных данных метода используются множество изменений для классификации $\Delta = \{\delta_r\}$, множество экспертных классов $C = \{c_i\}$ $(1 \le i \le n)$, набор метрик изменений μ, число кластеров k, которое первоначально задается равным числу экспертных классов n.

Выходные данные метода – множество классифицированных изменений. Опишем этапы метода, соответствующие блокам схемы, приведенной на рис. 6.

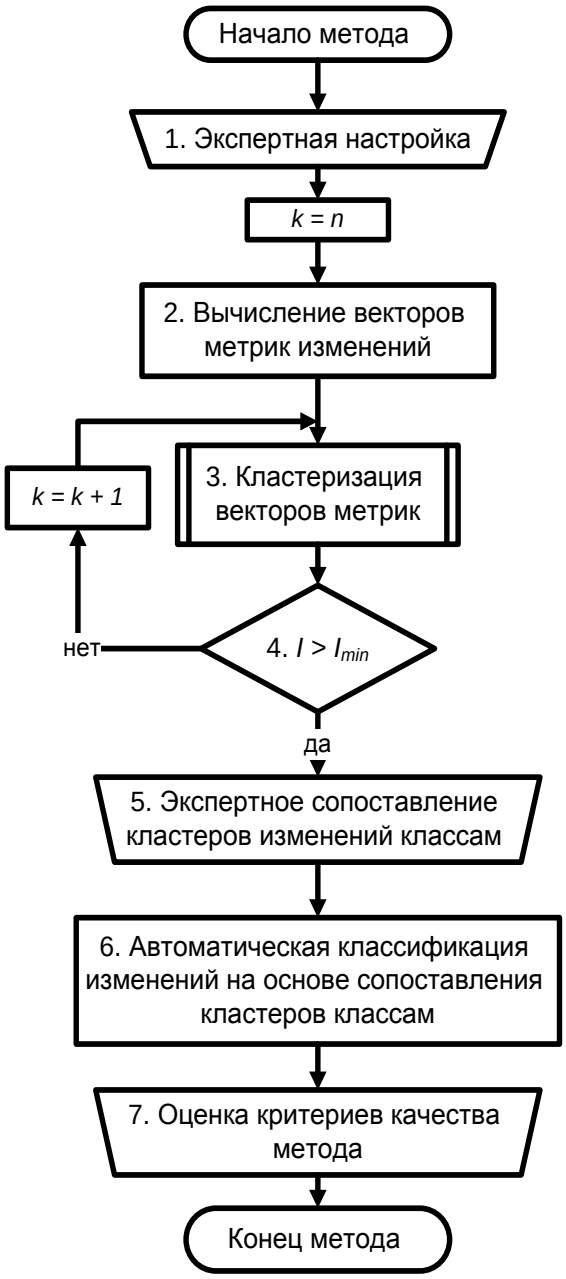

Рис. 6. Схема метода классификации изменений исходного кода на основе кластеризации метрик изменений

2.2.1. Экспертная настройка

В процессе настройки экспертом выбирается *набор метрик* [29, 23, 6, 2] *изменений исходного кода*, который позволяет обеспечить заданные уровни

P_{Cmin}, E_{Cmax} критериев качества классификации. Описание этих критериев приводится в пунктах 2.2.4, 2.2.7.

В четырех из пяти экспериментах, описанных в главе 3, используются одиннадцать специально выбранных метрик, приведенных в табл. 10, приводящих к удовлетворительным значениям характеристик качества для практического использования метода.

В других экспериментах использовалось от трех до нескольких десятков метрик [13, 15, 18]. При этом точность работы метода оказывалась недостаточной при использовании трех метрик. С другой стороны, в экспериментах с использованием большого числа (более двадцати) разнородных метрик также происходило снижение качества работы метода. В частности, при включении в набор метрик из табл. 10 метрик числа добавленных, измененных и удаленных строк комментариев в процессе кластеризации начинают выделяться кластеры по признаку добавления или удаления комментариев к коду, а также модификации комментариев. Приведенные аспекты не являются существенными для разбиения изменений на классы: *добавление новой функциональности, исправление логики, рефакторинг, удаление функциональности, форматирование кода*. Поэтому указанные метрики не были включены в набор для данной классификации, но могут пригодиться, например, для построения следующей классификации: *добавление новой функциональности с комментариями, добавление новой функциональности без комментариев, исправление кода и комментариев, перевод кода в комментарий, удаление функциональности и комментариев*. Такая классификация может быть полезной для отслеживания добавления некомментированной новой функциональности, а также распространенной среди программистов практики перевода кода в комментарий.

2.2.2. Вычисление векторов метрик изменений

Для каждого изменения формируется вектор из значений метрик, выбранных на этапе настройки метода. Формулы для расчета значений метрик

изменений приведены ниже. На выходе этапа формируется множество векторов метрик изменений $\{\mu\delta_r\}$.

Назовем *метрикой изменения* δ_r *исходного кода* числовую величину $\mu\delta_r$, которая характеризует данное изменение. Эту метрику для кода S предлагается вычислять по формуле:

$$\mu\delta_r = M^{\delta}(S_{r-1}, S_r),$$

где M^{δ} – функция вычисления метрики изменения. Функция M^{δ} вычисляется на основе кода, предшествующего изменению S_{r-1}, и измененного кода S_r. На практике более удобно производить расчет метрик в терминах добавленных, измененных и удаленных строк кода. Обобщим понятие строки кода l как последовательности лексем $s_1 s_2 \ldots s_e$, где s_e – лексема конца строки.

Каждое изменение δ_r можно представить в виде совокупности добавленных L_+, удаленных L_- и измененных L^* строк кода:

$$\delta_r = \{L_+, L_-, L^*\},$$

где L^* – множество пар $<l^*_-, l^*_+>$, где l^*_- – строка до внесения изменения и l^*_+ – измененная строка. В настоящей работе такое представление изменений производится методом поиска *редакционного предписания* [21].

Редакционное предписание – это последовательность действий по вставке, удалению и замене символов, необходимая для получения простейшим образом из одной строки другой. Например, для двух строк «ABC» и «ABDE» можно построить таблицу преобразований, приведенную в табл. 9.

Таблица 9. Преобразование строки «ABC» в строку «ABDE»

Операция по преобразованию	Совпадение	Совпадение	Замена	Вставка
Строка 1	A	B	C	
Строка 2	A	B	D	E

Пусть l_1 и l_2 — две строки (длиной n и m соответственно) над некоторым алфавитом, тогда редакционное расстояние $d(l_1$ и $l_2)$ можно подсчитать по следующей рекуррентной формуле:

$$d(l_1, l_2) = \begin{cases} 0; & i = 0, j = 0 \\ i; & j = 0, i > 0 \\ j; & i = 0, j > 0 \\ \min(& \\ \quad D(i, j-1) + 1, & \\ \quad D(i-1, j) + 1, & \\ \quad D(i-1, j-1) + m(l_1[i], l_2[j]) & \\ \quad); & i > 0, j > 0, \end{cases}$$

где *m(a, b)* равна нулю, если $a = b$ и единице в противном случае; *min(a, b, c)* возвращает наименьший из аргументов.

Рассмотрим формулу более подробно. Здесь шаг по i символизирует удаление из первой строки, по j — вставку в первую строку, а шаг по обоим индексам символизирует замену символа или отсутствие изменений. Очевидно, что редакционное расстояние между двумя пустыми строками равно нулю. Так же очевидно то, что чтобы получить пустую строку из строки длиной i, нужно совершить i операций удаления, а чтобы получить строку длиной j из пустой, нужно произвести j операций вставки. В нетривиальном случае нужно выбрать минимальную «стоимость» из трёх вариантов. Вставка/удаление будет в любом случае стоить одну операцию, а вот замена может не понадобится, если символы равны — тогда шаг по обоим индексам бесплатный. Формализация этих рассуждений приводит к формуле, указанной выше.

Алгоритм поиска редакционного предписания имеет сложность порядка $\Theta(n^2)$ времени и $\Theta(n^2)$ памяти, либо $\Theta(n^3)$ времени и $\Theta(n)$ памяти в зависимости от реализации [7].

Редакционное предписание используется для расчета некоторых метрик изменений. В данной работе редакционное предписание рассчитывается для файлов исходного кода с помощью системы контроля версий *Subversion* [91].

Метрики, используемые в диссертации при анализе исходного кода [29, 23, 6], приведены в табл. 10.

Таблица 10. Метрики изменений кода, используемые в диссертации

Обозначение метрики	Описание метрики	Обозначение метрики	Описание метрики
1. LOC_+	Число добавленных строк кода	*7.* F^*	Число измененных файлов исходного кода
2. LOC_-	Число удаленных строк кода	*8.* I_+	Число добавленных интерфейсов
3. LOC^*	Число измененных строк кода	*9.* I_-	Число удаленных интерфейсов
4. CC_+	Цикломатическая сложность добавленного кода	*10.* CS_+	Число добавленных классов и структур
5. CC_-	Цикломатическая сложность удаленного кода	*11.* CS_-	Число удаленных классов и структур
6. CC^*	Цикломатическая сложность измененного кода		

Приведем описание расчета указанных метрик изменений.

Метрики изменений LOC_+, LOC_-, LOC^*, основанные на числе строк кода, рассчитываются по следующим формулам:

$$LOC_+ = ||L_+||,$$
$$LOC_- = ||L_-||,$$
$$LOC^* = ||L^*||,$$

где $||A||$ – мощность множества A.

Расчет метрик изменений CC_+, CC_-, CC_*, основанных на метрике цикломатической сложности кода, приведен ниже.

Цикломатическая сложность кода предназначена для оценивания сложности потока управления программы [29, 23].

Зададим функцию *CC(l),* позволяющую вычислить упрощенную цикломатическую сложность строки кода l как число конструкций языка, управляющих потоком исполнения программы, которые встречаются в строке l:

$$CC(l) = \sum_{s_i \in \boldsymbol{l}} [s_i \in S_{CF}],\ 1 \le i \le n_l.$$

Здесь $\{s_i\}$ – совокупность лексем исходной строки l, n_l – число лексем в строке l, S_{CF} – множество лексем, представляющих конструкции управления потоком исполнения программы. Выражение $[B]$ в настоящей работе принимает значение *единица,* когда значение предиката B *истинно,* и *ноль,* когда значение B *ложно.*

Приведем формулы для расчета метрик цикломатической сложности добавленного CC_+, удаленного CC_-, и измененного кода CC^*:

$$CC_+ = \sum_{l_+ \in L_+} CC(l_+),$$
$$CC_- = \sum_{l_- \in L_-} CC(l_-),$$
$$CC^* = \sum_{<l_-^*, l_+^*, > \in L^*} (CC(l_+^*) - CC(l_-^*)).$$

Метрики CC_+, CC_- рассчитываются по цикломатическим числам добавленной и удаленной строк соответственно. Цикломатическая сложность CC^* рассчитывается как разность цикломатических чисел строки до внесения изменения и измененной строки.

Метрика числа измененных исходных файлов F^* рассчитывается с помощью встроенных возможностей системы контроля версий, которая позволяет запрашивать множество измененных файлов SF^* исходного кода в δ_r:

$$F^* = ||SF^*||.$$

Метрики числа добавленных I_+ и удаленных I_- интерфейсов рассчитываются по следующим формулам:

$$I_+ = \sum_{l_+ \in L_+} I(l_+) + \sum_{<l_-^*, l_+^*, > \in L^*} (I(l_+^*) - I(l_-^*)),$$
$$I_- = \sum_{l_- \in L_-} I(l_-) + \sum_{<l_-^*, l_+^*, > \in L^*} (I(l_-^*) - I(l_+^*)),$$

где $I(l)$ – число объявленных интерфейсов в строке кода – рассчитывается по следующей формуле:

$$I(l) = \sum_{s_i \in \boldsymbol{l}} [s_i = S_I],\ 1 \le i \le n_l.$$

Здесь S_I – лексема, представляющая собой конструкцию объявления интерфейса.

Метрики числа добавленных CS_+ и удаленных CS_- классов и структур рассчитываются по аналогичным формулам:

$$CS_+ = \sum_{l_+ \in L_+} CS(l_+) + \sum_{<l_-^*, l_+^*, > \in L^*} (CS(l_+^*) - CS(l_-^*)),$$
$$CS_- = \sum_{l_- \in L_-} CS(l_-) + \sum_{<l_-^*, l_+^*, > \in L^*} (CS(l_-^*) - CS(l_+^*)),$$

где *CS(l)* – число объявленных интерфейсов в строке кода – рассчитывается по следующей формуле:

$$CS(l) = \sum_{s_i \in \boldsymbol{l}} [s_i = S_C \vee s_i = S_S],\ 1 \le i \le n_l.$$

Здесь S_C – лексема, представляющая собой конструкцию объявления класса, а S_S – лексема, представляющая собой конструкцию объявления структуры.

Для каждого изменения δ_r рассчитывается p-мерный вектор метрик, где p – число используемых метрик:

$$\boldsymbol{\mu}\delta_r = <\mu_1\delta_r, \mu_2\delta_r, \ldots, \mu_p\delta_r>.$$

Векторы метрик изменений используются для распределения изменений по кластерам.

2.2.3. Кластеризация векторов метрик изменений

На данном этапе формируется множество кластеров изменений Q. Кластеры $q_j \in Q$ – это непересекающиеся подмножества изменений, объединенные по заданным критериям сходства в процессе кластеризации [5, 25].

Кластеризацию будем выполнять с помощью алгоритма *k-средних* [5, 25] с использованием *косинусной меры* [61] близости изменений.

Выбор алгоритма кластеризации *k-средних* объясняется простотой его использования при приемлемом качестве результата. При его использовании необходимо предположение относительно ожидаемого числа кластеров. На первом этапе метода число кластеров полагается равным числу экспертных классов, а в дальнейшем может уточняться.

Метод *k*-средних имеет недостаток – зависимость результата кластеризации от масштабов изменения отдельных переменных [5]. При использовании евклидовой меры близости изменения группируются в кластеры на основании их масштаба, а не экспертного класса.

Автором диссертации было найдено решение проблемы – использование меры близости изменений кода, не учитывающей длину векторов метрик изменений. Благодаря использованию меры ρ, основанной на косинусе угла между векторами метрик, появилась возможность выделения классов изменений без учета их масштаба:

$$\rho(v_1, v_2) = \cos(v_1, v_2),\ \cos(v_1, v_2) = \frac{v_1^t v_2}{\|v_1\|\|v_2\|},$$

где v_1, v_2 – векторы метрик изменений, v_1^t – транспонированный вектор v_1. Данная мера широко применяется при решении других задач, и оправдала себя, например, для кластеризации текстовых документов. Объекты считаются тем более близкими, чем ближе значение соответствующей меры ρ к единице.

Выбор метода кластеризации *k*-средних с косинусной мерой обосновывается достижением достаточных для практического использования значений критериев качества классификации. Подтверждающие это эксперименты приведены в разд. 3.2 третьей главы.

В алгоритме кластеризации векторов метрик изменений используются следующие входные данные: число кластеров *k* и векторы метрик изменений $\mu\delta_r$. Выходные данные алгоритма: множество кластеров изменений *Q*. Схема алгоритма приведена на рис. 7.

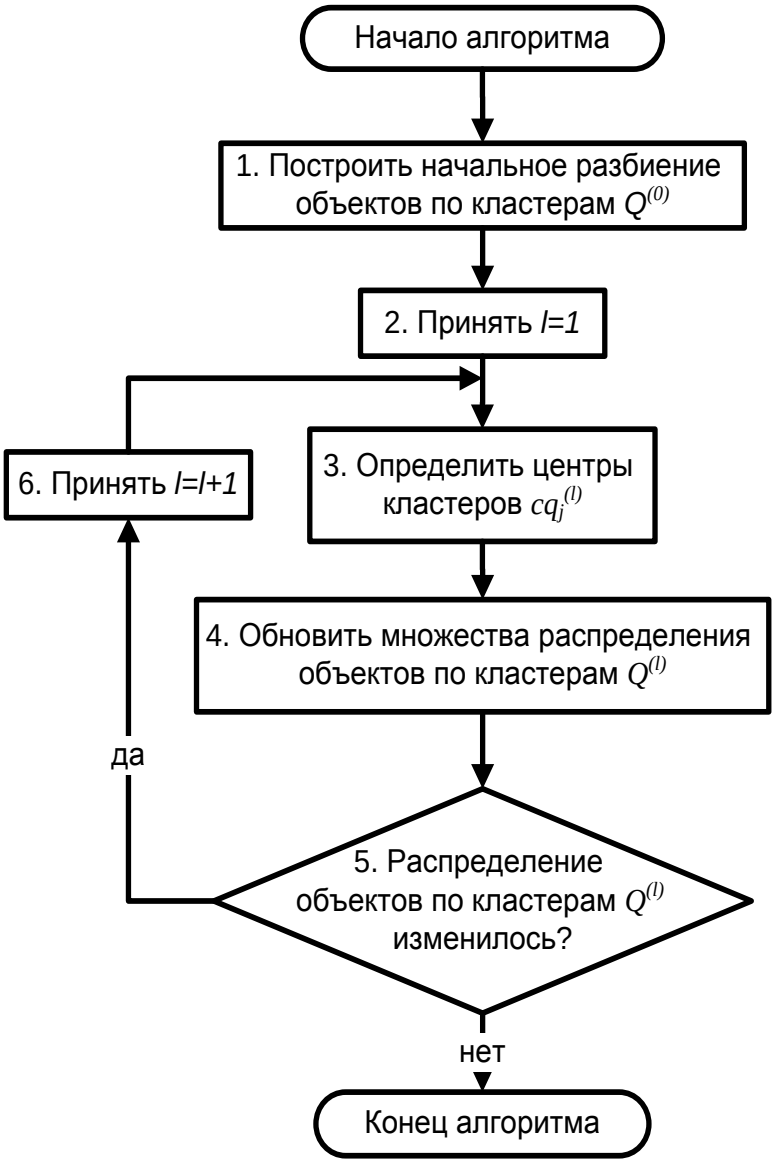

Рис. 7. Схема алгоритма кластеризации *k*-средних

Ниже приведено описание работы этого алгоритма [5]:

2.2.3.1. Производится начальное разбиение $Q^{(0)} = \{q_1^{(0)}, q_2^{(0)}, \ldots, q_k^{(0)}\}$ множества объектов $\{\delta_i\}$ таким образом, чтобы k наиболее далеко расположенных друг от друга изменений были первыми включены в различные кластеры:

$$q_1^{(0)}=\{\delta_1\},\ q_j^{(0)} = \{\delta_i \mid \rho(\mu\delta_i, cq_t^{(0)}) = min_{i,t}\,\rho(\mu\delta_i, cq_t^{(0)})\},$$

$$2 \le j \le k,\ 1 \le i \le N,\ 1 \le t \le j,$$

где N – общее число изменений.

2.2.3.2. Номер итерации l принимается равным единице.

2.2.3.3. Центры кластеров $cq_j^{(l)} = \{cq_j^{(l)}\}$ определяются по формуле:

$$cq_j^{(l)} = \frac{\sum_i \left([\delta_i \in q_j^{(l-1)}]\,\mu\,\delta_i\right)}{\sum_i [\delta_i \in q_j^{(l-1)}]},\ 1 \le i \le N,\ 1 \le j \le k,$$

2.2.3.4. Множества распределения объектов по кластерам обновляются по формуле $Q^{(l)} = \{q_j^{(l)}\}$, где:

$$Q^{(l)} = \{\delta_i \mid \rho(\mu\delta_i, cq_j^{(l)}) = max_i\,\rho(\mu\delta_i, cq_j^{(l)})\},\ 1 \le i \le N,\ 1 \le j \le k.$$

2.2.3.5. Проверяется условие: $\sum_i ||q_j^l - q_j^{l-1}|| = 0$. Здесь под знаком "–" понимается операция взятия симметрической разности множеств: $A–B = (A \cup B) \setminus (A \cap B)$. Если условие выполнено, то процесс завершается и осуществляется переход к шагу 3 и номер итерации l увеличивается на единицу.

Приведенный алгоритм позволяет автоматически разбить множество изменений на кластеры. В каждый из них объединяются наиболее схожие друг с другом изменения.

Этот алгоритм реализован в открытом программном средстве *CLUTO* [61]. Средство *CLUTO* используется в настоящей работе для кластеризации метрик изменений. Ниже приводится способ расчета функционалов качества кластеризации.

2.2.4. Оценка критериев качества кластеризации

Для оценки качества разбиения изменений на кластеры был выбран функционал I, как наиболее простая мера «плотности» группировки изменений по кластерам. Этот функционал рассчитывается по следующей формуле [61]:

$$I = \sum_{j=1}^{k} \sqrt{\sum_{\delta_1, \delta_2 \in q_j} \rho(\delta_1, \delta_2)},$$

где k – число кластеров, q_j – кластер с номером j, ρ – мера близости изменений, δ_1, δ_2 – изменения из кластера q_j. Смысл этого функционала состоит в том, что чем выше его значение, тем плотнее изменения группируются в пределах

кластеров, что свидетельствует о повышении качества решения задачи кластеризации.

На данном этапе метода проверяется условие превышения значения функционала I над заданным экспертом минимальным уровнем I_{min}. Если условие не выполняется, то проводится подбор нового числа кластеров k.

Другими критериями качества кластеризации [61] могут служить среднее значение близости изменений в пределах j-го кластера $M_I(q_j)$, его стандартное отклонение $\sigma_I(q_j)$, а также среднее значение близости изменений данного кластера к другим кластерам $M_E(q_j)$, а также его стандартное отклонение σ_E. Значения данных критериев используются для оценки близости изменений в рамках одного кластера, а также для оценки близости изменений каждого кластера к другим кластерам. Значения указанных критериев рассчитываются для множества значений меры близости пар изменений внутри кластера $\boldsymbol{\rho_I}$, а также для множества значений меры близости пар изменений между кластерами $\boldsymbol{\rho_E}$:

$$\boldsymbol{\rho_I}(q_j) = \{\rho(\delta_1, \delta_2) \mid \forall\, \delta_1, \delta_2 \in q_j, \delta_1 \neq \delta_2\},$$
$$\boldsymbol{\rho_E}(q_j) = \{\rho(\delta_1, \delta_2) \mid \forall\, \delta_1 \in q_j, \forall i, \delta_2 \in q_i, i \neq j\}.$$

Значения критериев рассчитываются по следующим формулам [61]:

$$M_I(q_j) = M\boldsymbol{\rho_I}(q_j),$$
$$\sigma_I(q_j) = \sqrt{D\boldsymbol{\rho_I}(q_j)},$$
$$M_E(q_j) = M\boldsymbol{\rho_E}(q_j),$$
$$\sigma_E(q_j) = \sqrt{D\boldsymbol{\rho_E}(q_j)},$$

где MX – математическое ожидание $x \in X$, а DY - дисперсия $y \in Y$.

2.2.5. Экспертное сопоставление кластеров изменений классам

На данном этапе каждому кластеру сопоставляется некоторый экспертный класс. При таком сопоставлении устанавливается соответствие

между кластерами q_1, q_2, …, q_k и классами изменений c_1, c_2, …, c_n. В настоящей работе предлагается проводить сопоставление кластеров классам на основании выборочной экспертной классификации изменений из каждого кластера и визуального анализа представления кластеров.

Визуальный анализ производится на основе трех способов визуального представления данных, которые строятся с помощью программного средства *CLUTO* [61, 84]. Это, во-первых, *график кластеров*, приведенный на рис. 8. Во-вторых, это *матрица кластеров*, которая приведена на рис. 9. В-третьих, это *матрица изменений*, фрагмент которой приведен на рис. 10.

График кластеров строится на основе многомерного шкалирования [66, 67], с помощью которого строится двумерное представление на основе исходного многомерного массива данных. Наиболее информативный признак кластера на графике – это его положение по отношению к другим кластерам. Расстояние между парами пиков на графике соответствует степени близости двух кластеров $M_E(q_i,q_j)$. Таким образом, кластеры, наиболее близкие в смысле используемой меры для кластеризации будут иметь соответствующие пики, расположены ближе всех друг к другу на графике. Высота каждого пика на графике пропорциональна среднему значению близости изменений в рамках соответствующего кластера $M_I(q_j)$. Объем пика пропорционален числу изменений внутри соответствующего кластера. И наконец, цвет пика представляет собой среднеквадратическое отклонение значений близости объектов внутри кластера $\sigma_I(q_j)$. Красный цвет представляет собой низкое его значение, тогда как голубой представляет собой высокое значение. Такое представление необходимо для выделения хорошо структурированных данных. Например, несложно идентифицировать кластеры с высокой степенью близости изменений путем поиска наиболее высоких пиков. Также легко идентифицировать кластеры с невысоким среднеквадратическим отклонением близости изменений путем поиска пиков красного или оранжевого цветов. Кластеры с высоким значением отклонения часто зашумлены и отображаются синим цветом.

Матрица кластеров строится с помощью представления нормированных значений метрик их центров с помощью цветов. Красному цвету соответствуют наибольшие значения метрик, зеленому – наименьшие.

Матрица изменений строится с помощью представления нормированных значений метрик изменений с помощью цветов. Кодировка значений цветов аналогична принятой для матрицы кластеров. В матрице изменений изменения группируются по кластерам.

В каждом кластере выделяется несколько изменений, наиболее близких к его центру. Производится их экспертная классификация. На основе преобладания изменений некоторого класса c в кластере q все объекты кластера q классифицируются как c: $q \to c$.

Процесс автоматизированной классификации изменений можно считать завершенным, если значения критериев качества, рассчитанных с помощью приведенных ниже формул, удовлетворяют заданным уровням. Иначе необходимо повторно выполнить этап экспертной настройки, описанный в п. 2.2.1.

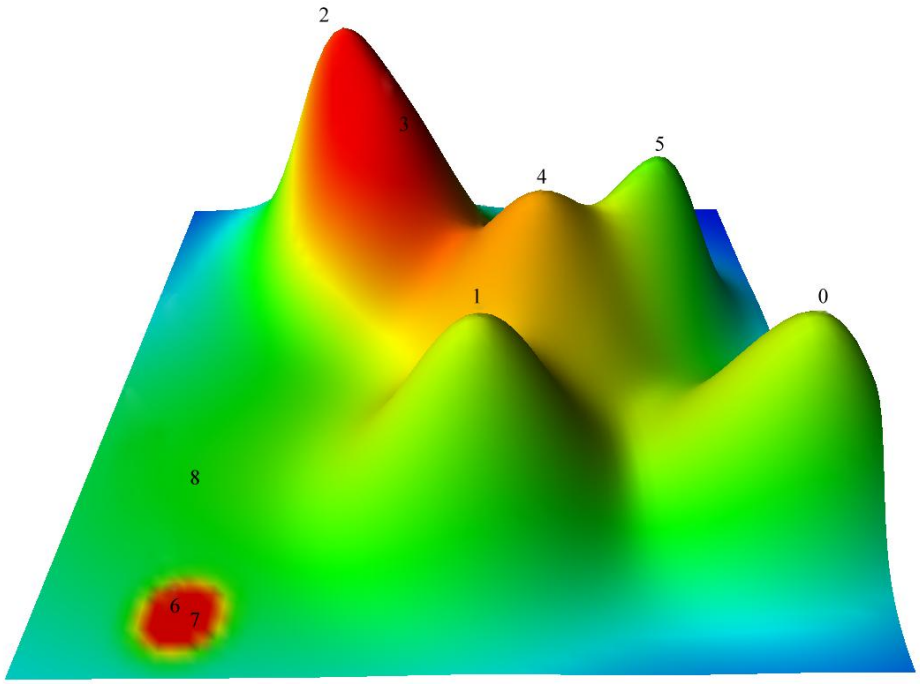

Рис. 8. График кластеров проанализированных изменений системы *Subversion.* Цифрами указаны соответствующие пикам номера кластеров

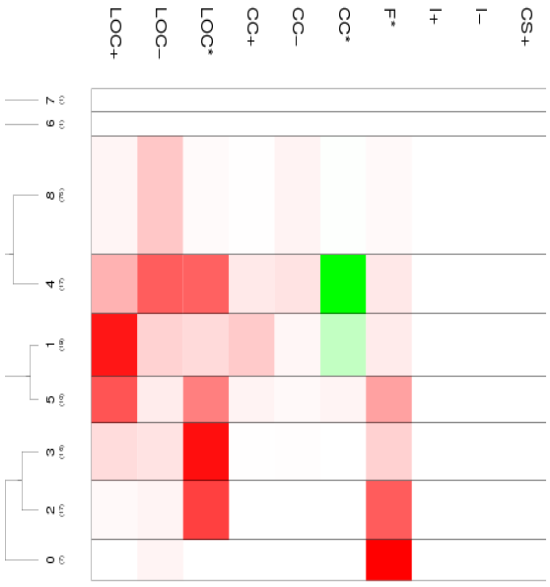

Рис. 9. Матрица кластеров, построенная для проанализированных изменений *Subversion.* По вертикали указаны номера кластеров, по горизонтали – названия метрик

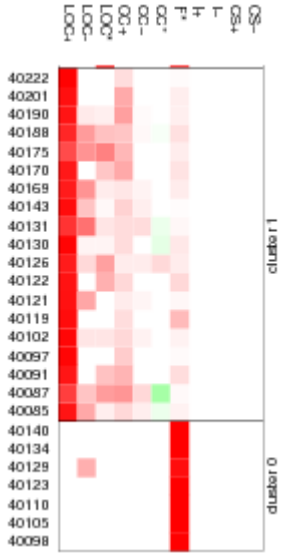

Рис. 10. Матрица изменений, построенная для кластеров 0 и 1 для проанализированных изменений *Subversion.* По вертикали слева указаны номера изменений (ревизии), справа – номера кластеров, по горизонтали – названия метрик

2.2.6. Автоматическая классификация изменений на основе сопоставления кластеров классам

Результатом автоматической классификации изменений на основе сопоставления содержащих их кластеров классам, является построенная классификация изменений по заданным экспертным классам.

2.2.7. Оценка критериев качества метода

Оценка качества метода автоматизированной классификации проводится на проверочном множестве изменений Δ_e.

2.2.7.1. Согласованность экспертной и автоматизированной классификации

Для оценки согласованности автоматизированной и экспертной классификации в работе используется *коэффициент Кохена* [42, 50].

Коэффициент Кохена представляет собой меру согласия, с которой два классификатора конкурируют в своих сортировках N элементов по n взаимно исключающим категориям. Классификатора в данном контексте может представлять человек или множество людей, которые коллективно распределяют N элементов, или некоторый алгоритм, который распределяет элементы на основе некоторого критерия.

Коэффициент Кохена вычисляется следующим образом. В табл. 11 размером $(n+1)x(n+1)$ выписываются результаты экспертной и автоматизированной классификации.

Таблица 11. Соответствие автоматизированной и экспертной классификации

Экспертная классификация	Автоматизированная классификация				
	c_1	c_2	…	c_n	Сумма
c_1					
c_2					
…			p_{ij}		p_i
c_n					
Сумма			p_j		$\sum p_{ij}$

Элемент табл. 11 p_{ij} – число изменений, отнесенных к классу c_i в ходе автоматизированной классификации и одновременно к классу c_j экспертной классификации. На главной диагонали расположены величины, равные количеству изменений, для которых совпадает результат автоматизированной и экспертной классификации. В строке *«Сумма»* приведено распределение частот изменений по типам автоматизированной классификации. В столбце *«Сумма»* приведено распределение частот изменений по экспертным типам. Правый нижний элемент табл. 11 содержит общее число анализируемых изменений.

Выражение для расчета коэффициента согласия Кохена следующее:

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

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 $\kappa < 1$. В общем случае, чем выше значение коэффициента κ, тем лучше согласие между классификаторами.

Вычисление коэффициента согласованности автоматизированной и экспертной классификации производится по кросс-табуляции наблюдаемого количества наблюдений изменений по каждому классу. Тогда величина p_α

рассчитываются как отношение суммы элементов табл. 11 по диагонали к общему числу элементов:

$$p_\alpha = \frac{\sum_{1 \le t \le n} p_{tt}}{\sum_{1 \le i,j \le n} p_{ij}}.$$

Величина p_e вычисляется как отношение суммы диагональных элементов таблицы, полученной случайным распределением по категориям, к общему числу элементов:

$$p_e = \sum_{1 \le k \le n} p_{k+}\, p_{+k},$$

где $p_{k+} = \frac{\sum_{1 \le t \le n} p_{kt}}{\sum_{1 \le i,j \le n} p_{ij}}$ и $p_{+k} = \frac{\sum_{1 \le t \le n} p_{tk}}{\sum_{1 \le i,j \le n} p_{ij}}$ – пропорции элементов по строкам и колонкам от общего их числа.

2.2.7.2. Чистота и энтропия разбиения изменений на кластеры и классы

Для каждого кластера q_j рассчитываются следующие критерии: чистота $P(q_j)$ и энтропия $E(q_j)$. Чистота $P(q_j)$ кластера q_j – отношение числа его изменений, принадлежащих экспертному классу c_i, соответствующему данному кластеру ($q_j \to c_i$), к числу изменений множества Δ_e в кластере q_j. Усредненные по всем кластерам значения чистоты P_Q и энтропии E_Q используются как оценки качества распределения изменений экспертных классов по соответствующим кластерам на всем множестве изменений Δ. Критерии чистоты P_Q и энтропии E_Q используются для проверки гипотезы, указанной выше.

Оценки соответствия автоматизированной и экспертной классификации предпочтительно строить в терминах экспертных классов с помощью альтернативных критериев чистоты P_C и энтропии E_C *распределения множества изменений по экспертным классам*. Их расчет проводится аналогично расчету значений P_Q и E_Q, с тем отличием, что роль кластеров в формулах играют экспертные классы. Если значения критериев P_C, E_C не

удовлетворяют заданным экспертом уровням P_{Cmin}, E_{Cmax}, то необходимо повторно выполнить этап экспертной настройки, описанный в п. 2.2.1.

Формулы для расчета значений чистоты $P_Q(\Delta_e)$ и энтропии $E_Q(\Delta_e)$ разбиения множества изменений Δ_e по кластерам Q следующие:

$$P_Q(\Delta_e) = \sum_{j=1}^{k} \frac{n_j^e}{N_e} P(q_j),$$

$$P(q_j) = \frac{n_j^a}{n_j^e},$$

$$E_Q(\Delta_e) = \sum_{j=1}^{k} \frac{n_j^e}{N_e} E(q_j),$$

$$E(q_j) = -\frac{1}{\log n} \sum_{i=1}^{n} \frac{n_j^i}{n_j^e} \log \frac{n_j^i}{n_j^e},$$

где N_e – общее число изменений множества Δ_e, n – число классов изменений, k – число кластеров, n^e_j – число попавших в кластер q_j изменений множества Δ_e, n^i_j – число попавших в кластер q_j изменений множества Δ_e, принадлежащих классу c_i, n^a_j – число попавших в кластер q_j изменений множества Δ_e, принадлежащих экспертному классу c^a_j которому сопоставлен кластер q_j. В случае, если n^i_j равно нулю, то значение члена $\frac{n_j^i}{n_j} \log \frac{n_j^i}{n_j} \frac{n_j^i}{n_j} \log \frac{n_j^i}{n_j}$ также полагается равным нулю.

Чистота $P(q_j)$ кластера q_j – отношение числа присутствующих в нем изменений экспертного класса, соответствующего этому кластеру, к общему числу классифицированных изменений кластера. Чистота характеризует степень соответствия кластера экспертному классу. Чем большее значение чистоты – тем лучше выполнена кластеризация.

Энтропия $E(q_j)$ кластера q_j характеризует неопределенность соответствия кластера экспертному классу. Чем меньше энтропия, тем лучше полученное решение. Чистота решения, в котором каждый кластер полностью представлен одним классом, равна единице, а энтропия – нулю.

Используем критерии P_Q, E_Q для проверки гипотезы, указанной выше. Для этого переформулируем ее в следующем виде:

«Для проверочного множества изменений Δ_e автоматизированный метод классификации изменений производит такое разбиение изменений по кластерам с последующем сопоставлением их классам, что для заданных уровней P_{Qmin}, E_{Qmax} справедливо следующее неравенство

$$P_Q(\Delta_e) \geq P_{Qmin}\,,\ E_Q(\Delta_e) \leq E_{Qmax}\,,$$

где P_{Qmin}, E_{Qmax} – допустимые уровни чистоты и энтропии распределения экспертных изменений по кластерам».

Идеальной кластеризации, когда в каждый кластер попадают изменения только одного экспертного класса, соответствует значение критерия P_{Qmin}, равного единице, и значение критерия E_{Qmax}, равного нулю.

Будем строить данные оценки с помощью следующих формул [19]:

$$P_{Qmin} = \mathrm{M}(P_Q) - u_{1-\alpha}\sigma(P_Q),$$
$$E_{Qmax} = \mathrm{M}(E_Q) + u_{1-\alpha}\sigma(E_Q),$$

где u_ε – ε-квантиль нормального распределения, α – уровень значимости, $M(X)$ – математическое ожидание X, $\sigma(Y)$ – среднеквадратическое отклонение Y.

Для проверки данной гипотезы и оценки доверительных интервалов значений чистоты и энтропии предлагается использовать известный метод размножения выборок, который применяется, так как в эксперименте недостаточно данных для статистического оценивания устойчивости метода. Классификация изменений – процесс, требующий существенных затрат времени экспертов. Поэтому на практике очень сложно построить множества классифицированных изменений достаточного объема.

2.2.7.3. Методы размножения выборок

Разделим заданное множество изменений на M равных частей. Затем исключим первую часть и используем M-1 оставшихся частей как отдельную выборку. Вернем исключенную на предыдущем шаге часть, исключим вторую

и используем оставшиеся части с номерами *1,3,…,M* как следующую выборку и так далее. На основе сгенерированных таким образом выборок произведем оценивание статистических параметров. Этот метод при *M = 1* называется методом складного ножа [20].

Существуют и другие методы размножения выборок, например, бутстреп [8, 28, 30, 34, 49]. Суть этого метода заключается в многократном случайном выборе изменений из исходной выборки для формирования любого заданного числа выборок.

Выводы по главе 2

Во второй главе сформулированы следующие положения:

1. Предложена гипотеза о возможности частичной автоматизации классификации изменений исходного кода методом кластеризации метрик. Предложен экспериментальный метод проверки гипотезы.
2. Разработан метод автоматизированной классификации изменений исходного кода на основе кластеризации метрик изменений, позволяющий сократить число изменений для классификации, выполняемой вручную, обладающий следующими достоинствами по сравнению с описанными в первой главе методами:
    - отсутствие узкоспецифичных эвристик;
    - независимость от языка программирования;
    - отсутствие необходимости построения обучающего множества;
    - адаптивность;
    - возможность настройки классификации с помощью метрик.
4. Теоретически обоснован выбор метода *k-средних* с *мерой близости* объектов для кластеризации, *основанной на косинусе угла между векторами метрик* изменений.

Глава 3. Применение автоматизированной классификации изменений в процессе разработки программного обеспечения

Наиболее важным активом проектов по разработке программных систем является исходный код системы. Большинство современных проектов хранит всю историю изменений исходного в хранилищах *систем контроля версий программного кода*. Каждое изменение, хранящееся в системе контроля версий, имеет множество атрибутов, среди которых общими для многих реализаций систем контроля версий [46, 85] являются *идентификатор изменения или ревизия* и *комментарий к изменению,* написанный на естественном языке.

Хранение изменений исходного кода в системе контроля версий открывает широкие возможности для анализа истории изменений программной системы. Анализ истории программной системы позволяет получить информацию о закономерностях процесса развития программной системы в прошлом, которую можно использовать для улучшения процесса развития в будущем.

Предлагаемый в настоящей работе метод автоматизированной классификации изменений программного кода позволяет повысить производительность работы участников команды разработки за счет частичной автоматизации экспертизы изменений исходного кода.

Экспертиза исходного кода – полезная практика [75, 76], состоящая в просмотре исходного кода на предмет поиска ошибок и проблем дизайна кода [75, 77, 78]. Эта практика позволяет выявить и разрешить большое количество проблем на ранней стадии разработки, пока исправление еще не требует больших затрат времени на интеграцию изменения и его тестирование.

Экспертизу кода можно разделить на две категории: *формальные инспекции и неформальную экспертизу*.

Под формальными инспекциями понимается экспертиза, проводимая, специально обученными людьми, которые ищут дефекты согласно строго

определенному процессу [82]. К формальным методам экспертизы исходного кода можно отнести, например, *инспекции Фэгана* [80].

Неформальная экспертиза кода не подчиняется строго определенному процессу, и может являться составной частью процесса разработки, например, в виде следующих форм:

- *экспертиза кода в парах*, когда автор кода и еще один человек вдвоем просматривают код;
- *экспертиза кода по запросу*, который автоматически может присылать, например, система контроля изменений, при появлении в ней нового изменения кода;
- *парное программирование* [77, 78], при котором весь процесс написания кода происходит в парах;
- *экспертиза кода с помощью автоматизированных средств*, когда используются программные средства для проверки типичных ошибок в коде.

Например, для выявления ошибки в программном коде, приведенном на ниже, скорее всего, будет достаточно, чтобы кто-либо другой, кроме автора, внимательно просмотрел этот код. Автор при экспертизе собственного кода может не заметить своей ошибки, так как он помнит свой код таким, каким его написал, поэтому важно, чтобы экспертиза кода выполнялся не тем человеком, который данный код разработал.

В коде, приведенном ниже

```
//…
for (int i = 0; i < m; ++i)
{
  for (int j = 0; i < n; ++j)
  {
    //…
  }
}
//…
```

ошибка заключается в том, что в условии завершения внутреннего цикла перепутана переменная цикла (*i* вместо *j*). Исправление такой ошибки на этапе

тестирования обычно ведет к бо́льшим затратам времени, чем на ранней стадии при экспертизе кода.

Еще один пример исходного кода с ошибкой:

```
public class LatLonFormatter
{
  private readonly int precision;
  public LatLonFormatter(int precision)
  {
    this.precision = precision;
  }
  public string FormatLat(double lat)
  {
    return string.Format(/*…*/);
  }
  public string FormatLon(double lon)
  {
    return string.Format(/*…*/);
  }
}

public class Vessel : LatLonFormatter
{
  private readonly double lat;
  private readonly double lon;
  public Vessel(double lat, double lon)
    : base(3)
  {
    this.lat = lat;
    this.lon = lon;
  }
  public override string ToString()
  {
    return FormatLat(lat) + ", " + FormatLon(lon);
  }
}
```

В этом примере некорректно используется наследование. Вместо отношения наследования между классами *Vessel* и *LatLonFormatter* здесь должно применяться отношение использования, поскольку класс *Vessel не должен использоваться как класс LatLonFormatter*. Эта ошибка может быть найдена только при экспертизе данного кода. Исправленный вариант приведен ниже:

```
public class LatLonFormatter
{
  public static string FormatLat(double lat, int
        precision)
```

```
    {
      return string.Format(/*…*/);
    }
    public static string FormatLon(double lon, int
          precision)
    {
      return string.Format(/*…*/);
    }
  }

  public class Vessel
  {
    private readonly double lat;
    private readonly double lon;
    public Vessel(double lat, double lon)
    {
      this.lat = lat;
      this.lon = lon;
    }
    public override string ToString()
    {
      return LatLonFormatter.FormatLat(lat, 3) + ", " +
          LatLonFormatter.FormatLon(lon, 3);
    }
  }
```

Существует множество исследований [75, 78, 80], подтверждающих факт, что экспертиза исходного кода, как в форме инспекций, так и в неформальном виде, позволяет экономить значительные средства при разработке программной системы за счет устранения ошибок на ранних фазах разработки программной системы.

Например, одна из организаций (*SmartBear Software* [89]) провела исследование [75], в котором на проекте размером в *10000* строк кода из десяти разработчиков в течение трех месяцев применялась экспертиза кода в парах. При сравнении стоимости исправления ошибок, найденных с использованием экспертизы исходного кода и в процессе тестирования и эксплуатации исходного кода выяснилось, что использование экспертизы исходного кода позволяет сократить расходы более чем в два раза: с *$368000* до *$152000*.

В работе *Project Management Body of Knowledge (PMBOK)* [35] (Американский национальный стандарт *ANSI/PMI 99-001-2004*) приводится следующий факт: способность участников проекта повлиять на конечные

характеристики продукта и окончательную стоимость проекта максимальны в начале проекта и уменьшаются по ходу выполнения проекта, как показано на рис. 11. Главная причина этого состоит в том, что стоимость внесения изменений в проект и исправления ошибок в общем случае возрастает по ходу выполнения проекта.

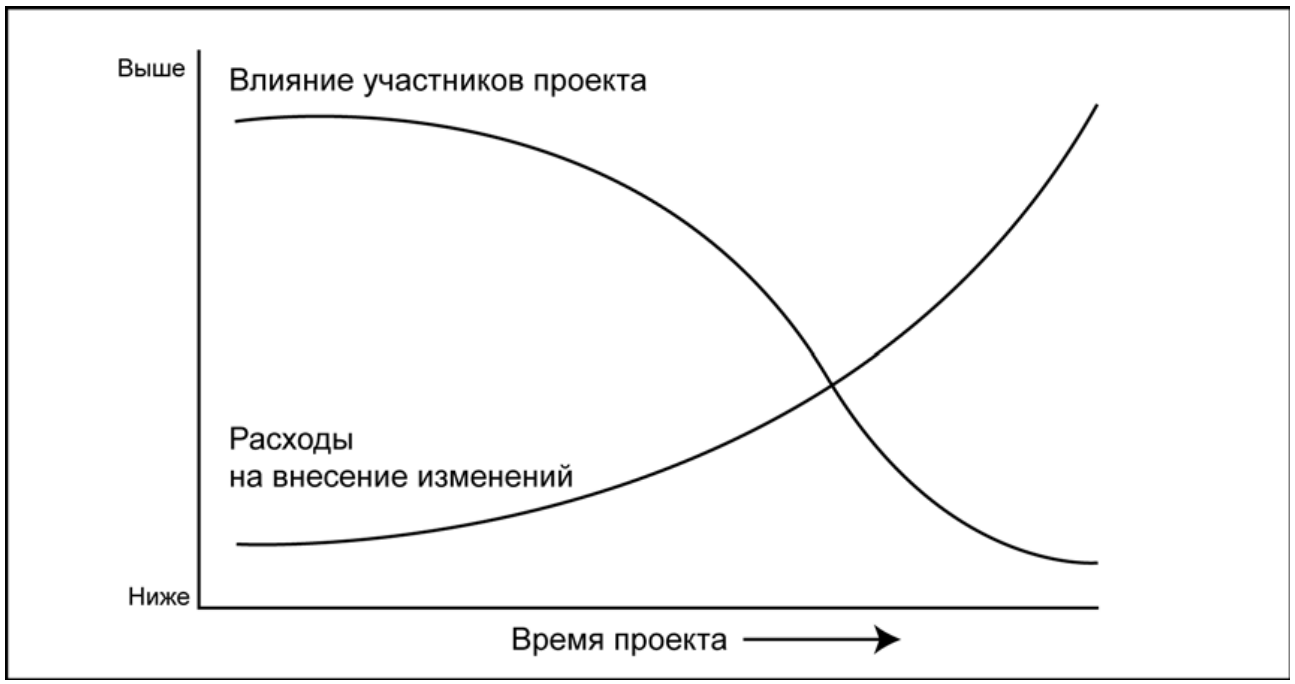

Рис. 11. Влияние участников проекта в течение проекта

Экспертиза программного кода действительно помогает устранять дефекты программных систем на ранних фазах разработки, пока затраты на их исправление еще не так велики, как на финальных фазах.

Все сказанное выше справедливо также для *экспертизы изменений программного кода,* получаемых с помощью системы контроля версий. Пример изменения исходного кода, отображаемого в виде построчного сравнения исходного и измененного кода с помощью специального инструмента *TortoiseMerge,* входящего в состав *TortoiseSVN* [93] системы контроля версий *Subversion* [91], приведен на рис. 12.

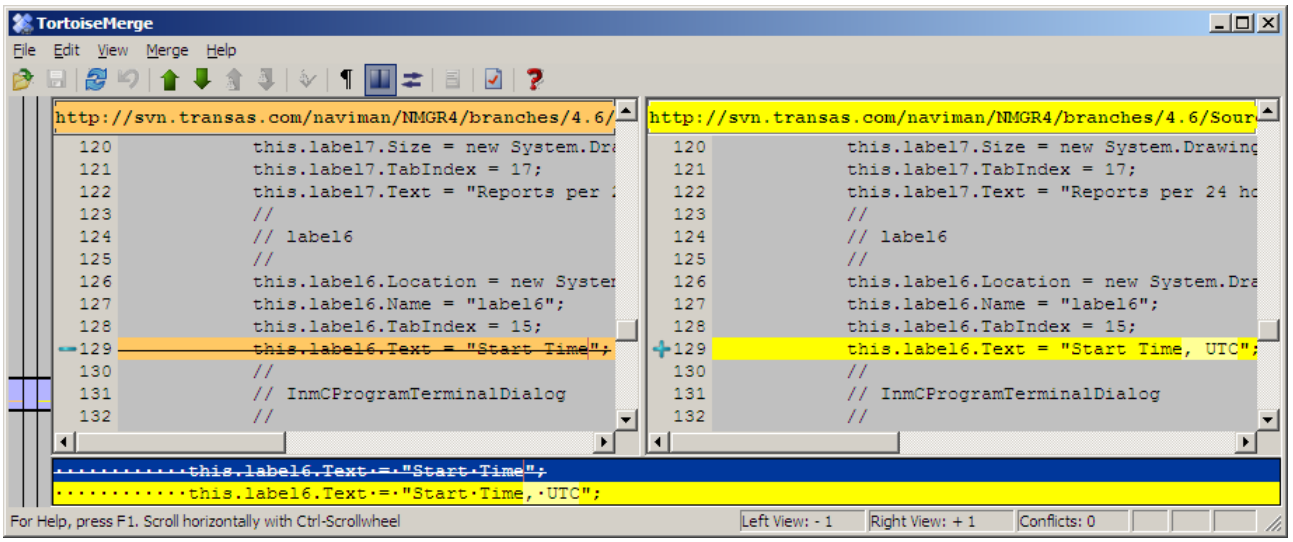

Рис. 12. Пример отображения изменения исходного кода с помощью инструмента *TortoiseSVN*

Экспертиза исходного кода программной системы вообще, и экспертиза изменений исходного кода в частности – *трудоемкая процедура*, которая влечет *дополнительные затраты на организацию процесса* в проекте разработки программной системы.

3.1. Использование автоматизированной классификации изменений в проекте разработки программной системы

Команда программистов может генерировать большое количество изменений, что может приводить к существенным затратам времени на их экспертизу. В табл. 12 приведены данные о числе изменений, внесенных в различные проекты за определенный период времени. В некоторых проектах число вносимых в исходный код изменений может быть очень большим.

Таблица 12. Число изменений исходного кода по проектам за месяц

Проект	*Subversion*	*Navi-Manager*	*LRIT*	*KDE*
Размер, *LOC*	~ 300 тысяч	~ 250 тысяч	~ 50 тысяч	~ 4.7 миллиона
Период наблюдения ~1 месяц	22.09.2007-22.10.2007		2.06.2008-2.07.2008	17.09.2007-14.10.2007
Количество изменений	442	72	400	11841 (!)

В табл. 12 приведены результаты подсчета количества изменений, внесенных в систему контроля версий за период приблизительно равный одному месяцу для проектов различного размера (*LOC* – число строк кода): системы контроля версий *Subversion* [91], клиент-серверной системы мониторинга мобильных объектов *Navi-Manager* [78] и компонент глобальной системы мониторинга флота *LRIT* [*70*], разрабатываемых автором данной статьи в компании ЗАО «Транзас Технологии», и графической оболочки для *Unix KDE* [83].

В разд. 3.3 покажем на различных вариантах применения в процессе разработки программной системы, что автоматизированная классификация изменений программного кода может быть полезной для повышения эффективности выполнения различных задач. Но сначала опишем модель организации процесса разработки программной системы, используемую в дальнейшем.

3.1.1. Модель организации программной системы

В настоящем разделе опишем модель организации проекта разработки программной системы, основанную на признанных в индустрии разработки программного обеспечения стандартах, разработках лидирующих организаций, и методологиях *(PMBOK* [35], *Object-Oriented Technology Center (IBM)* [57],

XP [78, 77] и др.), а также на базе проектов разработки некоторых программных систем (*Navi-Manager* [78]). Предлагаемая модель не претендует на общность, она лишь отражает общие черты многих проектов разработки программных систем.

Модель приведена в настоящей работе для иллюстрации на ней вариантов применения метода автоматизированной классификации изменений в процессе разработки программного продукта.

Деятельность по созданию программной системы обычно происходит в рамках *проекта разработки программной системы*.

Проект – это временное предприятие, предназначенное для создания уникальных продуктов, услуг или результатов [35].

Результатом проекта разработки программной системы является *программный продукт* в виде готовой программной системы или *услуга*, предоставляемая программной системой, или *результат*, полученный с помощью программной системы.

Проект разработки программной системы выполняется командой, каждый человек в которой действует согласно определенной *роли*. Роли участников проекта по разработке программной системы, в предлагаемой в настоящей работе модели организации процесса разработки, описаны ниже в настоящей главе.

Менеджеры проекта или организация могут разделить проект на фазы, чтобы обеспечить более качественное управление. Совокупность этих фаз составляет *жизненный цикл проекта* [35, 90]. Набор фаз жизненного цикла, в предлагаемой в настоящей работе модели организации процесса разработки, описан в параграфе «Фазы жизненного цикла проекта разработки программной системы» данного раздела настоящей главы.

3.1.1.1. Роли участников процесса разработки программной системы

Множество участников проекта разработки программной системы можно разделить по ролям, согласно которым они действуют. Диаграмма наиболее

распространенных ролей участников проекта разработки программной системы приведена на рис. 13.

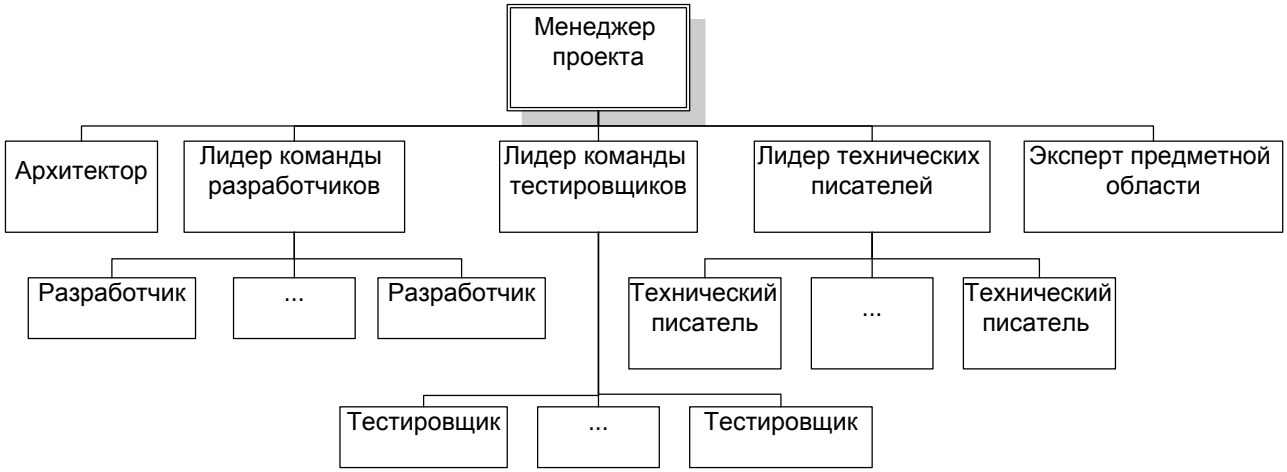

Рис. 13. Организационная диаграмма участников проекта разработки программной системы

Опишем основные обязанности участников процесса разработки программной системы, на основе ролевой структуры проекта по разработке программной системы, предложенной Центром объектно-ориентированной технологии компании *IBM* [57].

Разработчик программной системы занимается деятельностью, связанной с созданием исходного кода программной системы и обеспечением базового уровня надежности работы программной системы.

Технический лидер группы разработчиков формирует методики и планы разработки, участвует в проектировании программной системы, решает наиболее сложные технические проблемы, а также контролирует качество выполняемых подчиненными разработчиками работ и их производительность.

Архитектор программной системы отвечает за проектирование архитектуры системы, согласовывает развитие работ, связанных с проектом.

Тестировщик программной системы занимается проверкой соответствия фактического функционирования программной системы заданному в спецификации, включая наличие и корректность основной функциональности, производительность и стабильность работы.

Лидер группы по тестированию занимается разработкой методик и планов тестирования, а также контролирует качество выполняемых подчиненными тестировщиками работ и их производительность.

Технический писатель (разработчик информационной поддержки) занимается созданием документации для программной системы на основе предоставленных лидером разработчиков и менеджером проекта технических сведений относительно функционирования системы.

Лидер группы технических писателей формирует и согласует с менеджером проекта планы по созданию документации, получает у технического лидера команды разработчиков информацию о реализованной функциональности в программной системе, контролирует качество выполняемых подчиненными техническими писателями работ и их производительность.

Эксперт предметной области отвечает за изучение сферы приложения, поддерживает направленность проекта на решение задач данной области.

Менеджер проекта разработки программной системы занимается общим руководством подчиненными, стремясь добиться создания качественной программной системы, отвечающей всем заявленным требованиям в срок.

Как будет показано далее в разд. 3.2, метод автоматизированной классификации изменений программного кода позволит автоматизировать некоторые задачи, выполняемые перечисленными участниками проекта разработки программной системы.

3.1.1.2. Фазы жизненного цикла проекта разработки программной системы

В жизненном цикле проекта разработки программной системы можно выделить следующие фазы: *анализ предметной области, анализ элементов программной системы, спецификация, разработка архитектуры программной системы, кодирование, тестирование, развертывание, документирование, обслуживание системы* [68, 24].

В зависимости от принятой в проекте методологии разработки, порядок выполнения данных фаз может варьироваться, фазы могут накладываться друг на друга. Например, в *водопадной методологии разработки* [68] эти фазы выполняются строго последовательно. В *итеративных методологиях разработки* [68], таких как *XP* [77, 78], *SCRUM* [26], данные фазы выполняются циклически, повторяясь в рамках каждой *итерации*.

Ограничимся в рамках настоящей работы рассмотрением следующих фаз: *разработка архитектуры*, *реализация функциональности, тестирование, развертывание, документирование и обслуживание программной системы*. Ниже опишем содержание каждой из приведенных фаз.

Разработка архитектуры программной системы. Архитектура – это абстрактное представление программной системы. В течение данной фазы создается такая архитектура, чтобы программная система, построенная по ней, удовлетворяла существующим требованиям, а также могла быть изменена для того, чтобы удовлетворять будущим требованиям. В течение данной фазы также разрабатываются интерфейсы между программными модулями, а также требования к аппаратному обеспечению и системному программному обеспечению [32].

Кодирование или создание исходного кода программной системы. Создание исходного кода программной системы – это фаза жизненного цикла разработки, в ходе которой разрабатывается и улучшается исходный код программной системы. Настоящая фаза включает действия, выполняемые разработчиками, такие как: *кодирование, модульное тестирование, отладка, рефакторинг* [32, 76] и т. д.

Можно выделять следующие *стадии фазы кодирования*, связанные со стабильностью программной системы [26, 24]:

- *Стадия подготовки рабочих прототипов (Альфа-прототипирование)*, в которой добавляется новая функциональность и выпускаются промежуточные *альфа-версии*.

- *Стадия подготовки стабильных прототипов (Бета-прототипирование)*, в которой происходит отладка и исправление ошибок, и выпускаются промежуточные *бета-версии*. Указанная стадия отмечается объявлением в проекте состояния остановки реализации новой функциональности (*stop code*) [90], при котором запрещается внесение любых изменений в коде, кроме исправления найденных ошибок. Эта стадия предназначена для стабилизации версии перед развертыванием в производственной среде. Часто стадия подготовки стабильных прототипов совмещается с фазой тестирования, что позволяет оперативно устранять найденные в ходе тестирования ошибки.
- *Стадия стабилизации продукта*, в которой все найденные в ходе тестирования ошибки уже устранены. В *стабильных выпусках версий* исправляются найденные в процессе эксплуатации пользователем в производственной среде критичные ошибки. В течение этой стадии проект переводится в состояние «заморозки» (*freeze code*) [90], в котором разрешено внесение изменений только для исправления критичных ошибок. В качестве контроля может быть обязателен экспертиза каждого изменения хотя бы одним членом команды, кроме автора изменения, перед внесением его в систему контроля версий. Обычно в течение всего времени эксплуатации у заказчика развернутая версия программной системы находится в стабильной стадии. Из этого следует, что любое изменение, вносимое на данной стадии, должно строго контролироваться, как было описано выше.

Тестирование программной системы. Тестирование программной системы – это процесс, имеющий целью достижение должного уровня *качества программного обеспечения* [22]. Тестирование программного обеспечения представляет собой эмпирическое техническое расследование, выполняемое тестировщиком, призванное обеспечить заинтересованные стороны

информацией о качестве программного продукта или службы, находящихся в процессе тестирования, со ссылкой на контекст, в котором они оперируют [74]. Тестирование включает в себя, но не ограничивается выполнением программы с намерением найти ошибки в программном обеспечении. В ходе тестирования состояние и поведение программной системы в процессе работы сравнивается со спецификацией.

Развертывание программной системы. По завершении фазы тестирования стабильная версия программной системы развертывается в производственной среде. В результате выполнения этой фазы пользователь начинает эксплуатацию развернутой версии программной системы.

Документирование программной системы. В течение данной фазы создания программной системы членами команды создаются следующие виды документации: документация архитектуры, техническая документация, документация пользователя и другие разновидности. Если в процессе разработки программной системы выдвигается требование соответствия архитектурной, технической, а иногда и пользовательской документации текущему состоянию программной системы, тогда ее авторам необходимо постоянно следить за изменениями в программном коде и отражать их в документации.

Обслуживание программной системы. В случае если в программной системе, работающей в производственной среде, в процессе эксплуатации потребовалась доработка, тогда потребуется снова провести все фазы жизненного цикла для интеграции доработки в программную систему. Однако, уже на ранних фазах жизненного цикла понадобится учитывать, что продукт находится в стабильной стадии фазы кодирования, и с особой тщательностью контролировать внесение изменений, чтобы не нарушить стабильность системы.

Как уже было сказано выше, фазы жизненного цикла проекта разработки программной системы могут накладываться друг на друга. В дальнейшем нас будут интересовать в первую очередь *стадии фазы кодирования*, возможно,

совмещаемые с другими фазами жизненного цикла проекта. Каждая фаза кодирования ограничивает действия разработчиков программной системы определенным набором действий. Метод автоматизированной классификации изменений программного кода позволяет обеспечить контроль вносимых изменений на текущей стадии фазы кодирования, а также автоматизировать несколько типичных задач, выполняемых участниками проекта разработки в течение различных фаз жизненного цикла разработки программной системы.

Остановимся далее на описании применения метода автоматизированной классификации изменений и роли экспертизы исходного кода в процессе разработки программной системы.

Информация, выделенная путем анализа истории развития исходного кода, доступна только тем участникам проекта, которые технически подготовлены для анализа исходного кода, то есть 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, система контроля версий исходного кода содержит массу мелких и незначительных изменений, которые осложняют поиск важных, с точки зрения анализирующего, изменений.

Автоматизированная классификация изменений, в качестве вспомогательного инструмента, помогает выделить из исходного кода существенную информацию для анализирующего при выполнении задач, связанных с анализом истории программных систем. В частности, использование автоматизированной классификации позволяет отфильтровать несущественные с точки зрения 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 классов изменений на определенных стадиях фазы кодирования. Например, настроить инструмент автоматизированной классификации изменений так, чтобы при внесении изменения по реализации новой функциональности на стабильной стадии кодирования формировалось автоматическое уведомление о недопустимом изменении для технического лидера и автора данного изменения.

В настоящей главе также приводится несколько вариантов использования информации, выделенной из исходного кода путем автоматизированной классификацией изменений исходного кода другими участниками проекта. Тестировщику автоматизированная классификация изменений дает возможность получать информацию об изменениях, в которых реализуется новая функциональность, с помощью которых исправляются ошибки, в виде исходного кода или текста комментария, ассоциированного с изменением. Менеджеру проекта метод классификации изменений позволит строить отчеты с распределением изменений по классам.

3.1.2. Использование автоматизированной классификации изменений в проекте разработки программной системы

Автоматизированная классификация изменений может быть полезной всем членам команды разработки. Ниже подробно описаны возможные варианты использования инструмента, реализующего автоматическую классификацию изменений программного кода всеми участниками проекта [10, 13, 16, 17**Error! Reference source not found.**, 18].

3.1.2.1. Применение метода разработчиком

Рядовой разработчик программного обеспечения часто сталкивается с необходимостью экспертизы большого числа изменений. Например, при подключении к проекту, который уже имеет некоторую историю или по возвращению 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 погружения в детали каждого изменения. Ему достаточно выбрать набор классов изменений, который его интересует, и просмотреть изменения, соответствующие данному классу. На рис. 14 изображена схема экспертизы изменений выбранного класса.

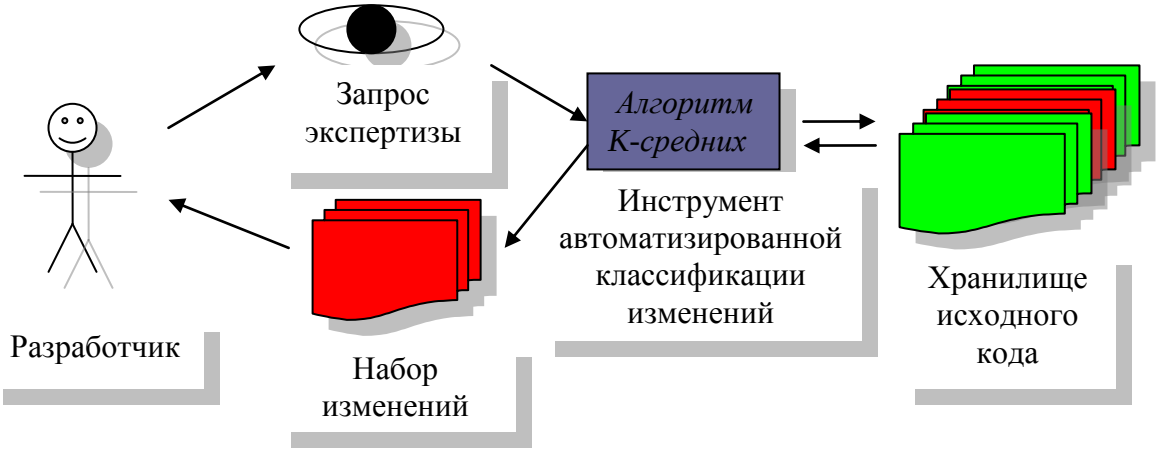

Рис. 14. Экспертиза изменений выбранного класса

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

Например, такие классы изменений, как реализация новой функциональности, рефакторинг, исправление логики могут служить источниками появления новых ошибок в программном коде.

3.1.2.2. Применение метода архитектором

Архитектор программной системы занимается созданием и адаптацией архитектуры под текущие требования в процессе развития программной системы. Очень часто в процессе разработки возникает ситуация, когда первоначально задуманная архитектура перестает соответствовать

изменившимся требованиям к системе. Нередки также случаи, когда задуманная изначально архитектура оказывается неадекватной вследствие ошибки при проектировании архитектуры, когда, например, закладывается слишком большая гибкость частей системы, которые не будут меняться или, наоборот, недостаточный уровень абстракции там, где он требуется.

В описанных случаях потребуется спроектировать новую архитектуру и произвести адаптацию текущей архитектуры системы к ней – рефакторинг [32, 76].

Архитектор должен следить за процессом внесения изменений в архитектуру системы, чтобы представлять себе фронт ведущихся разработчиками работ и их направление. Для этого ему может быть полезен метод автоматизированной классификации изменений программного кода, позволяющий выделить изменения класса *рефакторинг* за интересующий архитектора период времени.

3.1.2.3. Применение метода техническим лидером

Задачами технического лидера команды разработчиков, которые будем здесь рассматривать, являются: во-первых, регулярная экспертиза исходного кода и, во-вторых, контроль изменений, вносимых на текущей стадии разработки.

Рассмотрим задачу экспертизы исходного кода техническим лидером. Как уже было сказано, одной из основных мер поддержания качества кода на высоком уровне является постоянная экспертиза изменений, вносимых в код разработчиками.

Однако команда программистов может генерировать большое количество изменений, что может приводить физической неспособности технического лидера произвести экспертизу всех изменений (табл. 12).

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 не быть уделено должное внимание. Это, в свою очередь, приводит к потере контроля над качеством продукта.

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

Так, например, если техническому лидеру необходимо проанализировать только важные изменения, не связанные с исправлением логики, необходимо произвести экспертизу следующих классов изменений: *реализация новой функциональности*, *рефакторинг* и *удаление кода*.

Рассмотрим задачу контроля техническим лидером изменений, вносимых на текущей стадии фазы кодирования. Автоматизированная классификация изменений позволяет автоматизировать процесс контроля внесения изменений на текущей стадии процесса кодирования. Достаточно предоставить информацию о допустимых на текущей стадии кодирования изменениях. На рис. 15 показано возможное устройство модуля по обнаружению нежелательных изменений.

Технический лидер задает список классов изменений, запрещенных на текущей стадии кодирования. Затем регулярно производится автоматизированная классификация новых изменений из хранилища исходного кода. В случае, если класс изменения находится в списке запрещенных, автоматически формируется соответствующее уведомление техническому лидеру и разработчику, являющегося автором изменения.

В частности, в течение стадии стабилизации прототипа и стабилизации продукта стадии от разработчиков ожидается деятельность по исправлению ошибок в коде, а не реализации новых функций. Следовательно, в течение данных стадий в список запрещенных изменений будут внесены все классы изменений, за исключением класса «исправление логики».

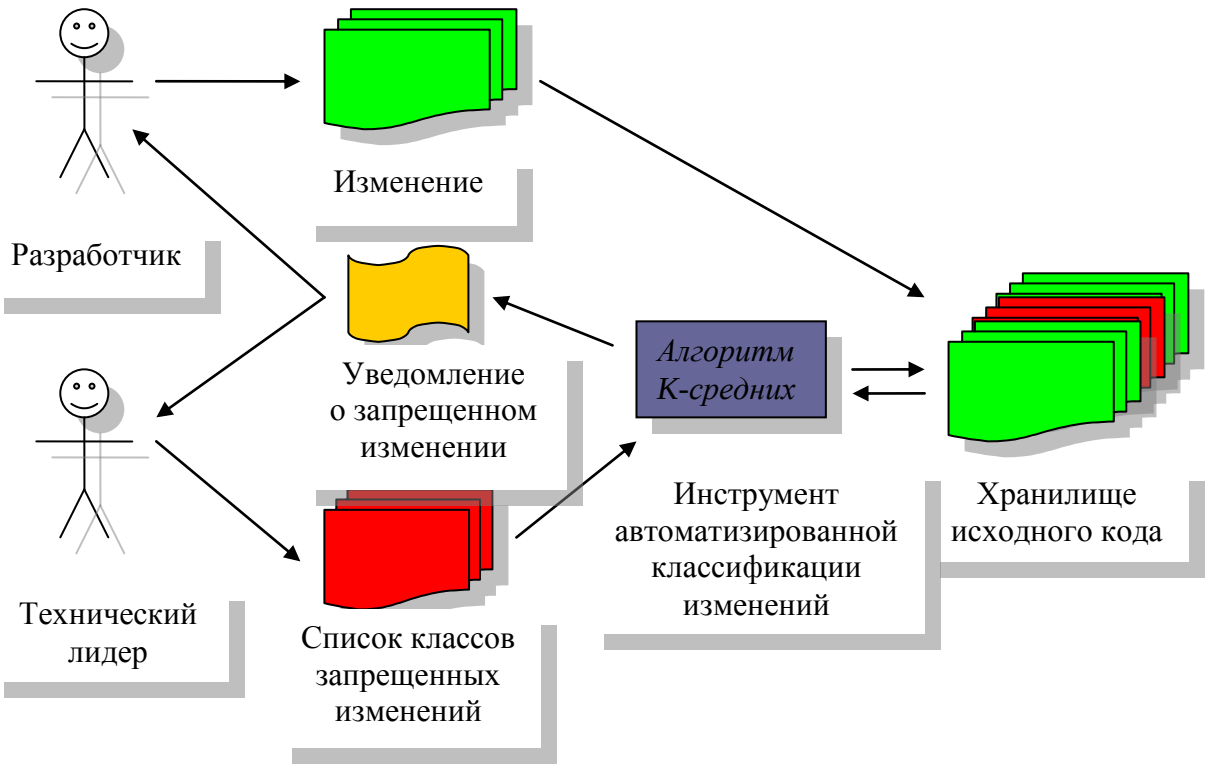

Рис. 15. Автоматизация поиска изменений, нежелательных на текущей фазе разработки

Кроме того, приведем в табл. 13 приведена допустимость классов изменений для описанных в разд. 3.1.1.2 стадий фазы кодирования, принятых в проекте *Navi-Manager*.

Таблица 13. Пример допустимости классов изменений для различных стадий кодирования

Стадия / Класс изменения	Альфа-прототипирование	Бета-прототипирование	Стабилизация продукта
Реализация новой функциональности	Допустимо	Не допустимо	Допустимо только для функциональности, проверенной в других версиях
Рефакторинг	Допустимо	Не допустимо	Не допустимо
Косметическое изменение	Допустимо	Не допустимо	Не допустимо
Исправление логики	Допустимо	Допустимо	Допустимо только для исправления критичных ошибок
Удаление функциональности	Допустимо	Допустимо	Допустимо в крайних случаях

На данном примере можно видеть, что на стадии разработки новой функциональности (*альфа-прототипирование*), очевидно, допустимы любые классы изменений. На стадии *бета-прототипирования* больше не разрешается реализация новой функциональности, не проводится рефакторинг и не делаются косметические изменения кода, чтобы не нарушить достигнутую стабильность системы. Разрешается только исправлять ошибки и удалять функциональность, которую не планируется включать в финальную *стабильную версию*. На стабильной стадии разрешены следующие классы изменений: перенос функциональности, проверенной в рамках других версий (ветвей разработки), исправление логики, причем, только для исправления критических ошибок (степень критичности определяет пользователь), удаление функциональности в крайних случаях (степень критичности определяет пользователь).

3.1.2.4. Применение метода специалистами по тестированию, лидером технических писателей и экспертом предметной области

В процессе работы над проектом лидеру специалистов по тестированию, лидеру технических писателей, а также эксперту предметной области приходится взаимодействовать с разработчиками для уточнения состояния проекта. Им часто не хватает информации относительно новой функциональности, реализованной в очередной версии программного обеспечения. Так, лидер по тестированию должен составить методику и план тестирования новых функций продукта. Эксперт предметной области должен как можно раньше изучить реализованную новую функциональность, чтобы при необходимости внести предложения по ее изменению для адаптации к нуждам пользователей. Лидер технических писателей должен распределить среди подчиненных задания по описанию новой функциональности в документации программной системы.

Иногда единственный достоверный способ выяснить полный список новых функций в конкретной версии программного обеспечения состоит в полной

экспертной классификации изменений кода за интересующий период или в доскональном изучении возможностей данной версии программного продукта. В этой ситуации применение автоматизированной классификации уменьшает время на проведение операции за счет отсеивания изменений, классифицирующихся как нечто кроме реализации новой функциональности.

3.1.2.5. 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. 16 приведено распределение изменений по классам для проекта *Navi-Manager* за период один месяц стадии реализации новой функциональности. По рис. можно сделать вывод, что проект *Navi-Manager* движется недостаточно быстро из-за того, что основные ресурсы команды разработчиков тратятся в основном на доработки существующего кода, а не на реализацию новой функциональности.

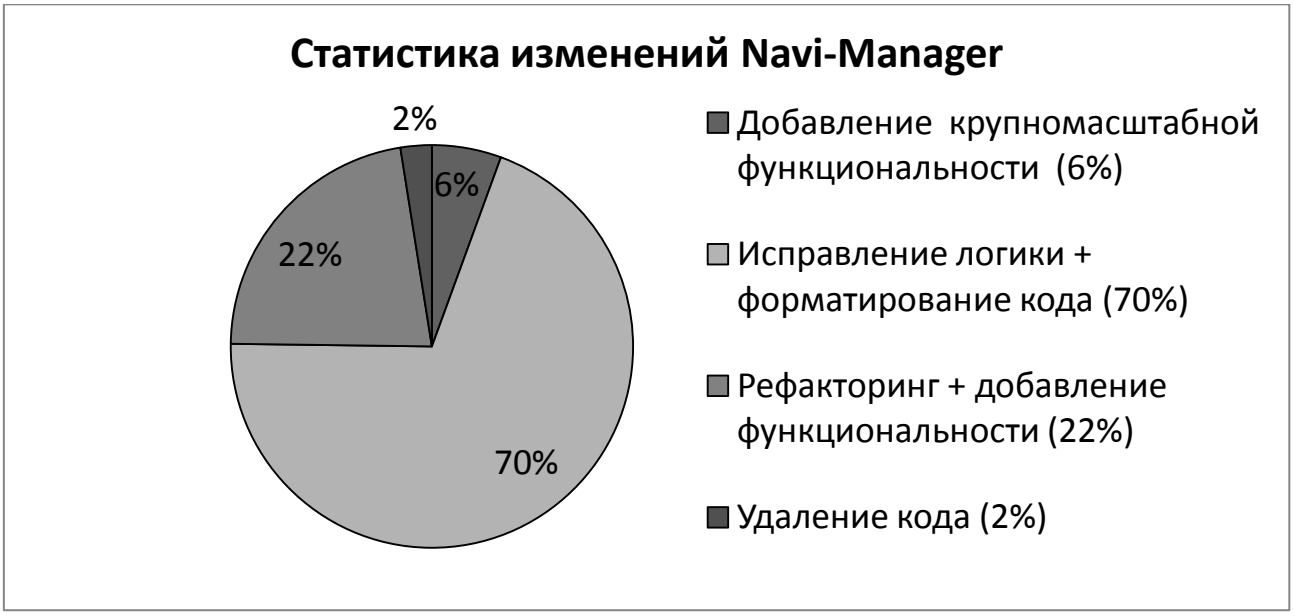

Рис. 16. Распределение изменений по классам в процессе разработки *Navi-Manager*

В данном разделе предложено использовать автоматизированную классификацию изменений программного кода для повышения эффективности

экспертизы исходного кода в проекте разработки программного продукта. Применение автоматизированной классификации действительно позволяет повысить эффективность экспертизы исходного кода, а также дает возможность автоматизации контроля изменений, вносимых на ответственных этапах разработки. Предлагается применять автоматизированную классификацию изменений программного кода для экспертизы важных изменений разработчиком и техническим лидером, для предоставления списка новой функциональности в конкретной версии продукта для лидера тестировщиков и технических писателей, списка выполненного рефакторинга для архитектора, списка исправлений ошибок для тестировщиков, а также построения отчетов распределения изменений по типам для менеджера проекта.

Опыт применения автоматизированной классификации изменений показывает его эффективность при решении задач экспертизы кода по сравнению с полной экспертизой изменений, а также позволяет получать дополнительную существенную информацию о процессе разработки и контролировать некоторые параметры.

Применение автоматизированной классификации исходного кода в проектах *Navi-Manager, LRIT, e-Tutor 5000* позволило оптимизировать процесс экспертизы изменений исходного кода, а также выявить проблему эффективности разработки проекта *Navi-Manager*. В процессе анализа проекта *Navi-Manager* наблюдался *значительный* уровень согласованности экспертной и автоматической классификации ($\kappa = 0.79$).

Итак, применение автоматизированной классификации изменений исходного кода способствует повышению эффективности экспертизы изменений кода при сохранении приемлемого для практического использования качества, а также предоставляет дополнительные механизмы контроля состояния процесса разработки.

Однако существует ряд ограничений применимости метода автоматизированной классификации на практике.

Если в процессе работы над проектом производится *небольшое количество изменений*, тогда нет необходимости в их автоматизированной классификации, так как организация полной экспертизы этих изменений не потребует большого количества времени.

В процессе применения метода на практике выявлена проблема классификации смешанных изменений, одновременно сочетающих в себе несколько модификаций различных классов. Планируется работа по корректной классификации таких изменений [11].

В следующем разделе описано внедрение в компании *ЗАО* «*Транзас Технологии*» автоматизированной классификации изменений исходного кода на основе кластеризации метрик, а также результаты применения метода для классификации изменений программных систем с открытым кодом.

3.2. Результаты использования метода

Внедрение метода автоматизированной классификации изменений выполнено (о чем свидетельствуют акты внедрения) при создании следующих программных систем в компании *ЗАО* «*Транзас Технологии*»:

1. Системы мониторинга мобильных объектов *Navi-Manager* [78].
2. Компонент системы глобального мониторинга флота *LRIT* [70].
3. Системы *e-Tutor 5000* [52] контроля действий студентов в процессе обучения на судовом [51], крановом [47] и других тренажерах [79, 69].

Предлагаемый метод использовался также при анализе программных систем с открытым кодом: системы контроля версий *Subversion* [91] и объектной обертки над реляционными базами данных *NHibernate* [80].

Опишем использование предложенного метода в процессе внедрения в указанных программных системах, доработки системы с открытым кодом *NHibernate* и анализа исходного кода открытой системы *Subversion*.

3.2.1. Внедрение метода при разработке системы *Navi-Manager*

В текущем разделе приводятся результаты применения метода в проекте разработки программной системы *Navi-Manager* [78]. *Navi-Manager* – это система мониторинга мобильных объектов, в том числе судов флота, состоящая из серверной и клиентской частей. Система преимущественно реализована на языке программирования *C#*. Общее число строк кода – более 200 тысяч.

Первоначальный выбор метрик для автоматизированной классификации осуществляется на основе эмпирических представлений эксперта о способности разделения экспертных типов изменений алгоритмом кластеризации по различиям значений их метрик. Выбранные метрики, предположительно связанные с различением экспертных классов изменений, приведены в табл. 14. Метрики анализируемых изменений рассчитывались как разности значений приведенных метрик для измененного кода и кода до внесения изменения.

Таблица 14. Метрики, используемые в процессе кластеризации

Метрика	Описание
eLOC	Эффективное число строк кода
CC	Цикломатическая сложность (число независимых путей в графе исполнения)
IC	Интерфейсная сложность (общее число параметров во всех методах)
C/S	Число классов и структур

Входные данные внедрения метода в проекте *Navi-Manager* за один месяц разработки приведены в табл. 15.

В процессе автоматизированной кластеризации было построено множество из четырех кластеров, которые были сопоставлены экспертом указанным в табл. 15 классам. При этом экспертом было классифицировано 18 изменений.

Таблица 15. Входные данные внедрения метода классификации изменений в системе *Navi-Manager*

Общая информация	
Экспертные классы C	*Добавление крупномасштабной функциональности, удаление кода, исправление логики + форматирование кода, рефакторинг + добавление функциональности*
Метод кластеризации метрик изменений	k-средних
Мера близости векторов метрик ρ	Евклидово расстояние между векторами метрик
Набор метрик для кластеризации μ	Приведен в табл. 14
Входные данные метода	
Общее число классифицируемых изменений	72
Число классифицированных экспертом изменений для сопоставления кластеров классам	18
Данные настройки метода	
Выбранное число кластеров k	4

Распределение множества из 72 изменений по кластерам приведено в табл. 16.

Таблица 16. Распределение изменений системы *Navi-Manager* по кластерам

Кластер	0	1	2	3
Число изменений	2	16	50	4

Распределение изменений по классам на основе экспертного сопоставления кластеров классам приведено в табл. 17. Для сопоставления

кластеров классам эксперту потребовалось выполнить классификацию 18 изменений.

Таблица 17. Распределение изменений системы *Navi-Manager* по экспертным классам

Класс	Добавление крупно-масштабной функциональ-ности	Исправление логики + форматиро-вание кода	Рефакторинг + добавление функциональности	Удаление кода
Сопоставленные кластеры	3	2	1	0
Число изменений	4	50	16	2

Для иллюстрации соответствия автоматизированной и экспертной классификации изменений случайным образом выбрано 13 изменений близкого масштаба за длительный период, и произведена экспертная классификация этих изменений. Результаты классификации приведены в табл. 18.

При анализе изменений программной системы *Navi-Manager* классификация изменений позволила выявить проблему процесса разработки, заключающуюся в большом числе ошибок, допущенных при программировании. При такой ситуации не следует экономить время на экспертизе изменений, так как сложившаяся ситуация требует контроля всех изменений кода, пока не будет решена указанная проблема.

Таблица 18. Результаты автоматизированной и экспертной классификации изменений из системы *Navi-Manager*. Обозначение экспертных классов: «Удал.» – удаление кода, «Реф.» – рефакторинг, «Испр.» – исправление логики, «Нов.» – новая функциональность.

Номер изме-нения	Комментарий	Класс изме-нения	Номер кластера
1	Перенос файлов между проектами, удаление ненужных	Удал.	0
2	Уменьшено дублирование кода	Удал.	
3	Обобщена логика *FindOrCreateUserSource*	Реф.	1
4	Рефакторинг работы с интервалами	Реф.	
5	Обобщена обработка *GlobeWireless*	Реф.	
6	Увеличен таймаут *PositionDiscarder* до 5 мин.	Испр.	2
7	Исправлено несколько логических ошибок в работе *TrackAgent*	Испр.	
8	Переименован *ITrackFetch.Interval* в *DestInterval*	Реф.	
9	Поправлена ошибка при проверке флагов рисования элемента меню	Испр.	
10	Поправлена установка *vessel.LastReportTime*	Испр.	3
11	Добавлены блоки try..catch в методы OnStart, OnStop сервиса	Нов.	
12	Добавлен перенос *max LastReportTime, LastSourceId* в логику объединения данных	Нов.	
13	Добавлена инверсия цвета текста в колонку *L* в *Vessel List View*	Нов.	

По результатам эксперимента получено значение коэффициента согласия Кохена κ = 0.79, которое свидетельствует о значительной степени согласованности двух классификаторов [50].

Использование метода автоматизированной классификации изменений в процессе разработки системы *Navi-Manager* позволило более эффективно решать ряд задач, связанных с контролем качества разработки кода и обменом информацией о производимых изменениях между участниками процесса, описанных в разд. 3.1 данной главы.

В результате использования метода в процессе разработки программной системы *Navi-Manager* обнаружился нежелательный эффект зависимости деления изменений на кластеры от их масштаба, отрицательно влияющий на качество результата работы метода. Это связано с особенностями используемого алгоритма кластеризации и используемой в данном эксперименте меры близости. Для последующих экспериментов применялась мера близости объектов кластеризации (изменений) на основе косинуса угла между векторами метрик, что позволило устранить нежелательный эффект.

3.2.2. Применение метода при доработке системы *NHibernate*

Программная система с открытым исходным кодом *NHibernate* использовалась в рамках разрабатываемой в компании *ЗАО «Транзас Технологии»* системы *Navi-Manager*. В процессе использования потребовалась доработка системы *NHibernate*. Ознакомиться с программным кодом системы, а также понять, в каком направлении движется разработка, позволил метод автоматизированной классификации изменений. Это позволило осознать необходимость доработки системы собственными силами, так как не наблюдалось активности по развитию требующейся функциональности.

Входные данные эксперимента по использованию предложенного метода для анализа кода системы *NHibernate* приведены в табл. 19.

Таблица 19. Входные данные эксперимента по использованию метода классификации изменений в системе *NHibernate*

Общая информация	
Экспертные классы C	*Удаление кода, реализация новой функциональности, исправление логики, рефакторинг, форматирование кода с изменением комментариев*
Метод кластеризации метрик изменений	k-средних
Мера близости векторов метрик ρ	Косинус угла между векторами метрик
Набор метрик для кластеризации $\boldsymbol{\mu}$	Приведен в табл. 10
Входные данные метода	
Общее число классифицируемых изменений	2069
Число размноженных изменений проверочного множества	272
Число классифицированных экспертом изменений для сопоставления кластеров классам	73
Данные настройки метода	
Выбранное число кластеров k	12

Опишем, как выполняется автоматизированная классификация изменений.

1. В процессе **экспертной настройки** выбраны метрики, приведенные в табл. 10. Проверочное множество Δ_{e68} из 68 изменений выбрано следующим образом. Из каждого выделяемого автоматизированным методом класса случайным образом выбрано по 16 изменений. Из 80 полученных отобрано 68 изменений, для которых совпали результаты экспертной классификации двух независимых экспертов.

2. Произведено **вычисление векторов метрик** для рассматриваемого множества из 2069 изменений.
3. Проведено разбиение векторов метрик изменений на двенадцать кластеров. Выбор числа кластеров k проводился на основе анализа значения функционала качества кластеризации I, а также размера «плохих» кластеров с сильно отличающимися друг от друга мерами близости изменений.
4. Осуществлена оценка критерия качества кластеризации для различного числа кластеров k от 5 до 15. Значение функционала качества кластеризации I увеличивается с ростом значений k. При значении $k = 12$ достигается наименьший размер самого «плохого» кластера – 547 изменений.
5. Сопоставление кластеров изменений экспертным классам проведено с помощью отбора изменений из каждого кластера, их экспертной классификации и установлении соответствия кластера классу на основе преобладания в нем изменений данного класса.
6. Автоматическая классификация изменений на основе сопоставления их кластеров классам проводится на основе предыдущего шага.
7. Значения критериев качества метода чистоты $P_Q(\Delta_{e68})$ и энтропии $E_Q(\Delta_{e68})$ разбиения проверочного множества Δ_{e68} по кластерам приведены в табл. 20.

Таблица 20. Расспределение изменений проверочного множества по кластерам. Обозначения экспертных классов: «И» – исправление логики, «Ф» – форматирование кода с изменением комментариев, «У» – удаление кода, «Н» – реализация новой функциональности, «Реф.» – рефакторинг, n^e_j – число изменений множества Δ_{e68} в кластере q_j

№ кластера j	Результат экспертного сопоставления $q_j \rightarrow c_i$	Экспертные классы c_i					n^e_j	P_Q	E_Q
		И	Ф	Н	У	Реф.			
0	И	9	0	0	1	1	11	0,82	0,37
1	Ф	1	4	0	0	0	5	0,8	0,31
2	Ф	0	1	0	0	0	1	1	0
3	Ф	0	0	0	0	0	0	1	0
4	Ф	1	7	0	0	0	8	0,88	0,23
5	Н	0	0	8	0	0	8	1	0
6	Н	0	0	4	0	0	4	1	0
7	Н	1	1	1	0	0	3	0,33	0,68
8	У	1	0	0	13	1	15	0,87	0,3
9	Реф.	2	1	0	0	3	6	0,5	0,63
10	Реф.	0	0	0	0	1	1	1	0
11	Реф.	2	2	0	0	2	6	0,33	0,68
Итого		17	16	13	14	8	68	0,78	0,32

Распределение множества из 2069 изменений по кластерам и экспертным классам приведено в табл. 21.

Таблица 21. Распределение изменений системы *NHibernate* по кластерам

Кластер	0	1	2	3	4	5	6	7	8	9	10	11
Число изменений	208	127	116	368	225	141	108	160	67	1	1	547

Распределение изменений по классам на основе экспертного сопоставления кластеров классам приведено в табл. 22.

Таблица 22. Распределение изменений системы *NHibernate* по экспертным классам

Класс	Форма-тирование кода и коммен-тариев	Реализация новой функциональ-ности	Исправле-ние логики	Рефакто-ринг	Удаление кода
Сопос-тавленные кластеры	2, 9, 10, 11	3, 4, 8	0	5, 6, 7	1
Число изменений	665	660	208	409	127

В процессе сопоставления кластеров классам экспертом было классифицировано 73 изменения, а также использовался визуальный анализ матрицы кластеров, приведенной на рис. 17 и графика кластеров, приведенного на рис. 18, сгенерированных программным средством *CLUTO*. В разд. 2.2.5 приведено описание способа построения этих матриц и графика.

Автоматизированная классификация предложенным в работе методом изменений позволила классифицировать 2069 изменений. При этом экспертом было классифицировано лишь 73 изменения.

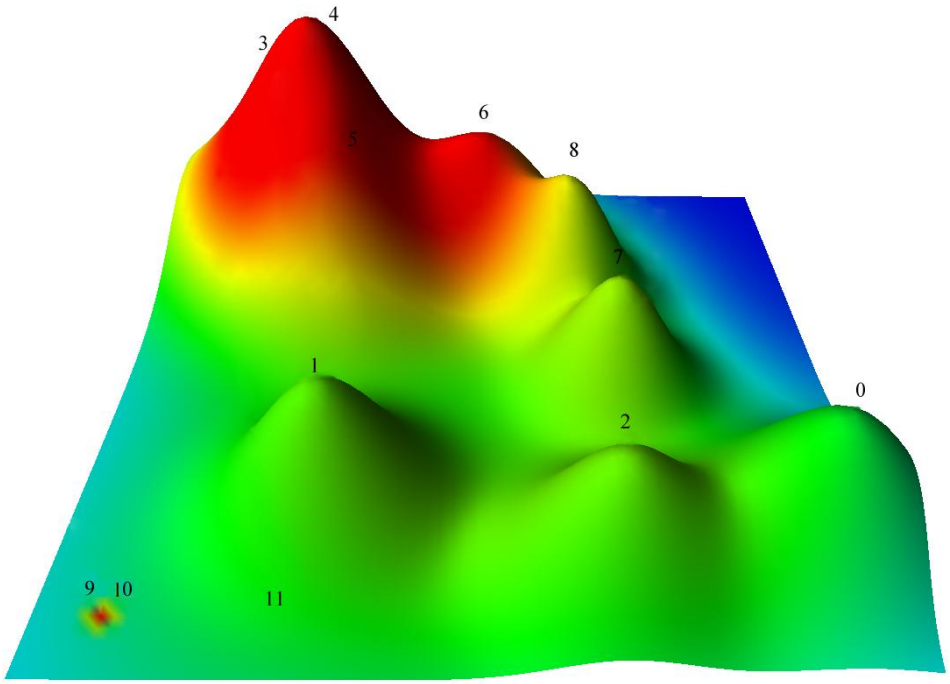

Рис. 17. График кластеров проанализированных изменений системы *NHibernate.* Цифрами указаны соответствующие пикам номера кластеров

Рис. 18. Матрица кластеров, построенная для проанализированных изменений *NHibernate.* По вертикали указаны номера кластеров, по горизонтали – названия метрик

В табл. 23 приведены результаты оценки критериев P_C и E_C. Значения в таблице получены, во-первых, объединением строк для кластеров, представляющих один и тот же экспертный класс, расчета $P_C(\Delta_{e68})$ и $E_C(\Delta_{e68})$ для них, и, во-вторых, усреднением полученных значений по пяти экспериментам, проведенным с помощью описанного во второй главе метода размножения выборок с параметром M = 13. Уровень значимости при построении доверительных интервалов для значений P_C и E_C принят равным 0,05.

Таблица 23. Усредненные значения критериев качества автоматизированной классификации изменений программной системы *NHibernate*

Число изменений размноженного проверочного множества	272
Значения критериев качества классификации	$P_C = 0{,}75 \pm 0{,}05$ $E_C = 0{,}37 \pm 0{,}06$

Из данной таблицы следует, что для проанализированных изменений программной системы в среднем 75% изменений корректно классифицированы разработанным методом, что приемлемо для практического использования метода. Кроме того, полученное значение энтропии свидетельствует о наличии существенной неопределенности соответствия кластеров классам. На этапе экспертного сопоставления кластера классам было обнаружено, что это в основном связано с недостаточно качественным выделением кластеров, соответствующих классу *рефакторинг*.

Применение автоматизированной классификации изменений позволило распределить 2069 изменений по пяти классам на основе кластеризации по двенадцати кластерам.

Для сравнения предлагаемого подхода с «чисто экспертной оценкой» приведем следующие цифры. Только 73 изменения были классифицированы для сопоставления кластеров классам в ходе экспертной классификации. Автоматизированный классификатор распределил 53 изменения проверочного множества из 68 по тем же классам, что и эксперты.

При анализе изменений программной системы *NHibernate* экспертом не рассматривалась существенная их часть, классифицированная как форматирование кода и комментариев (665 изменений). Это позволило выделить число существенных изменений системы, что позволяет отказаться от экспертизы 30 процентов остальных изменений. При необходимости анализа изменений, принадлежащих только классу *реализация новой функциональности*, экономия на экспертизе изменений составит около 65 процентов от 2069 изменений. При анализе изменений класса *исправление логики* экономия на экспертизе изменений составит уже более 85 процентов от 2069 изменений.

Следовательно, в результате применения метода **удалось существенно сократить затраты времени экспертов** на классификацию изменений.

На основе проведенных экспериментов можно сделать вывод, что метод работает хорошо, если производятся однотипные изменения, а когда в изменениях сочетаются разнородные модификации кода, то качество автоматизированной классификации ухудшается.

Из изложенного следует, что метод позволяет сократить время классификации изменений по сравнению с «чисто ручной» экспертизой при удовлетворительном для практики качестве.

3.2.3. Внедрение метода при разработке компонент системы *LRIT*

Система *LRIT* предназначена для идентификации и отслеживания положения судов в мировом масштабе. Компоненты *национальный центр данных* (*National Data Centre, NDC*), *провайдер сервиса приложений* (*Application Service Provider, ASP*), разрабатываемые в *ЗАО «Транзас Технологии»*, призваны обеспечить функционирование оборудования судов в рамках системы *LRIT* для отдельных государств.

В процессе разработки указанных компонент, предложенный в диссертации метод автоматизированной классификации изменений исходного

кода, применялся для повышения эффективности экспертизы кода и обмена информацией о его изменениях внутри команды разработки.

Входные данные внедрения за период один месяц разработки приведены в табл. 24.

Таблица 24. Входные данные эксперимента по внедрению метода классификации изменений в процессе разработки системы *LRIT*

Общая информация	
Экспертные классы C	*Удаление кода, реализация новой функциональности, исправление логики, рефакторинг, форматирование кода с изменением комментариев*
Метод кластеризации метрик изменений	k-средних
Мера близости векторов метрик ρ	Косинус угла между векторами метрик
Набор метрик для кластеризации μ	Приведен в табл. 10
Входные данные метода	
Общее число классифицируемых изменений	230
Число размноженных изменений проверочного множества	1892
Число классифицированных экспертом изменений для сопоставления кластеров классам	48
Данные настройки метода	
Выбранное число кластеров k	9

Распределение изменений по кластерам и экспертным классам приведено в табл. 25.

Таблица 25. Распределение изменений системы *LRIT* по кластерам

Кластер	0	1	2	3	4	5	6	7	8
Число изменений	16	15	11	6	62	2	19	23	76

В процессе сопоставления экспертом было классифицировано 48 изменений. Распределение изменений по классам на основе экспертного сопоставления кластеров классам приведено в табл. 26.

Таблица 26. Распределение изменений системы *LRIT* по экспертным классам

Класс	Форма-тирование кода и коммен-тариев	Реализация новой функциональ-ности	Исправле-ние логики	Рефакто-ринг	Удаление кода
Сопос-тавленные кластеры	8	2, 7	6	1, 3, 4, 5	0
Число изменений	76	34	19	85	16

Достигнутые в эксперименте значения критериев качества следующие: P_C = 0,71 ± 0,02, E_C = 0,43 ± 0,02. Эти значения получены на проверочном множестве, состоящем из 44 изменений с размножением выборки методом складного ножа (всего 1892 изменения).

Полученное значение энтропии объясняется наличием большого числа изменений, содержащих разнородные модификации. Это приводит к необходимости классифицировать изменений как принадлежащее наиболее вероятному возможному классу. Данная проблема может быть решена в рамках разработки модификации метода на основе нечеткой логики, как предлагается в разд. «Темы перспективных исследований».

Повышение эффективности экспертизы кода было достигнуто благодаря частичной автоматизации классификации изменений. Это позволило распределить 230 изменений на 5 классов. При этом экспертом было классифицировано лишь 48 изменений. Благодаря этому возможно контролировать качества кода, ограничиваясь экспертизой изменений интересующих эксперта классов. Это позволяет экономить на экспертизе от 40 до 70 процентов проанализированных изменений в зависимости от стадии разработки. Кроме того, это позволяет организовать обмен информацией об изменениях внутри команды разработки в виде различных отчетов.

3.2.4. Внедрение метода при разработке системы *e-Tutor 5000*

Универсальная система *e-Tutor 5000* предназначена для оценки компетенции студентов на тренажерах навигационной прокладки *Navigation Trainer Professional (NTPro) 5000*, тренажера машинного отделения *Engine Room Simulator (ERS) 5000*, кранового тренажера *Crane 5000*, нефтеналивного тренажера *Liquid Cargo Handling Simulator (LCHS) 5000* производства *ЗАО «Транзас Технологии»*. Система *e-Tutor 5000* на момент написания диссертации находится в процессе разработки.

В процессе разработки системы *e-Tutor 5000* предложенный в диссертации метод автоматизированной классификации изменений исходного кода применялся для повышения эффективности экспертизы кода и обмена информацией о его изменениях внутри команды разработки.

Входные данные эксперимента для 141 изменения за один месяц разработки приведены в табл. 27.

Таблица 27. Входные данные эксперимента по внедрению метода классификации изменений в процессе разработки системы *e-Tutor 5000*

Общая информация	
Экспертные классы C	*Удаление кода, реализация новой функциональности, исправление логики, рефакторинг, форматирование кода с изменением комментариев*
Метод кластеризации метрик изменений	k-средних
Мера близости векторов метрик ρ	Косинус угла между векторами метрик
Набор метрик для кластеризации $\boldsymbol{\mu}$	Приведен в табл. 10
Входные данные метода	
Общее число классифицируемых изменений	141
Число размноженных изменений проверочного множества	1980
Число классифицированных экспертом изменений для сопоставления кластеров классам	63
Данные настройки метода	
Выбранное число кластеров k	13

Распределение изменений по кластерам и экспертным классам приведено в табл. 28.

Таблица 28. Распределение изменений системы *e-Tutor 5000* по кластерам

Кластер	0	1	2	3	4	5	6	7	8	9	10	11	12
Число изменений	24	12	23	1	3	1	16	12	7	6	11	10	15

В процессе сопоставления экспертом было классифицировано 63 изменения. Распределение изменений по классам на основе экспертного сопоставления кластеров классам приведено в табл. 29.

Таблица 29. Распределение изменений системы *e-Tutor 5000* по экспертным классам

Класс	Форма-тирование кода и коммен-тариев	Реализация новой функциональ-ности	Исправле-ние логики	Рефакто-ринг	Удаление кода
Сопос-тавленные кластеры	0, 3, 5	2, 11	8, 9, 12	6, 7, 10	1, 4
Число изменений	26	33	28	39	15

Достигнутые в эксперименте значения критериев качества следующие: P_C = 0,73 ± 0,01, E_C = 0,46 ± 0,01. Эти значения получены на проверочном множестве, состоящем из 45 изменений с размножением выборки методом складного ножа (всего 1980 изменений).

Полученное значение энтропии объясняется наличием большого числа изменений, содержащих разнородные модификации.

Повышение эффективности экспертизы кода было достигнуто благодаря частичной автоматизации классификации изменений. Это позволило классифицировать 141 изменение на 5 классов. При этом экспертом было классифицировано 63 изменения. Благодаря этому возможно контролировать качества кода, ограничиваясь экспертизой изменений интересующих эксперта классов. Это позволяет экономить на экспертизе от 30 до 45 процентов проанализированных изменений в зависимости от стадии разработки. Кроме того, это позволяет организовать обмен информацией об изменениях внутри команды разработки в виде различных отчетов.

3.2.5. Использование метода при анализе истории разработки системы *Subversion*

Разработанный в диссертации метод применялся для анализа изменений системы контроля версий с открытым кодом *Subversion*. Система написана на языке программирования *C*. Известно, что процесс разработки Subversion хорошо структурирован, а изменения вносятся аккуратно, при этом большинство изменений можно четко классифицировать.

Входные данные эксперимента по использованию предложенного метода для анализа изменений кода системы *Subversion* за период две недели октября 2009 года приведены в табл. 30.

Таблица 30. Входные данные эксперимента по использованию метода классификации изменений в системе *Subversion*

Общая информация	
Экспертные классы *C*	*Удаление кода, реализация новой функциональности, исправление логики, рефакторинг, форматирование кода с изменением комментариев*
Метод кластеризации метрик изменений	*k*-средних
Мера близости векторов метрик ρ	Косинус угла между векторами метрик
Набор метрик для кластеризации μ	Приведен в табл. 10

Продолжение табл. 30

Входные данные метода	
Общее число классифицируемых изменений	163
Число размноженных изменений проверочного множества	2070
Число классифицированных экспертом изменений для сопоставления кластеров классам	32
Данные настройки метода	
Выбранное число кластеров *k*	9

Распределение изменений по кластерам и экспертным классам приведено в табл. 31.

Таблица 31. Распределение изменений системы *Subversion* по кластерам

Кластер	0	1	2	3	4	5	6	7	8
Число изменений	7	19	17	16	17	10	1	1	75

Распределение изменений по классам на основе экспертного сопоставления кластеров классам приведено в табл. 32.

Таблица 32. Распределение изменений системы *Subversion* по экспертным классам

Класс	Форматирование кода и комментариев	Реализация новой функциональности	Исправление логики	Рефакторинг
Сопос-тавленные кластеры	0, 6, 7, 8	1, 5	2, 3	4
Число изменений	84	29	33	17

В процессе сопоставления экспертом было классифицировано 32 изменения. В процессе сопоставления также использовался визуальный анализ матрицы кластеров, приведенной на рис. 19, матрицы изменений, приведенной на рис. 20, и графика кластеров, приведенного на рис. 21, сгенерированных программным средством *CLUTO*. В разд. 3.4 приведено описание способа построения этих матриц и графика.

Достигнутые в эксперименте значения критериев качества следующие: $P_C = 0{,}78 \pm 0{,}01$, $E_C = 0{,}35 \pm 0{,}02$. Эти значения получены на проверочном множестве, состоящем из 46 изменений с размножением выборки методом складного ножа (всего 2070 изменений).

Полученное значение энтропии, также как и в других экспериментах, объясняется недостаточным разделением классов изменений *рефакторинг* и *исправление логики*. Подбор метрик, позволяющих произвести более качественную классификацию таких изменений, является темой будущих исследований.

Автоматизированная классификация предложенным в работе методом изменений позволила классифицировать 163 изменения. При этом экспертом было классифицировано лишь 32 изменения.

При экспертном анализе изменений программной системы это позволило не рассматривалась существенную их часть, классифицированную как

форматирование кода и комментариев (84 изменения). В этот же класс попали изменения кода сопутствующих инструментальных средств, не затронувшие анализируемую часть программной системы. Таким образом, число изменений для экспертного анализа сократилось более чем на 30 процентов. При необходимости анализа изменений, принадлежащих только классу *реализация новой функциональности*, экономия на классификации изменений экспертом составит более 60 процентов.

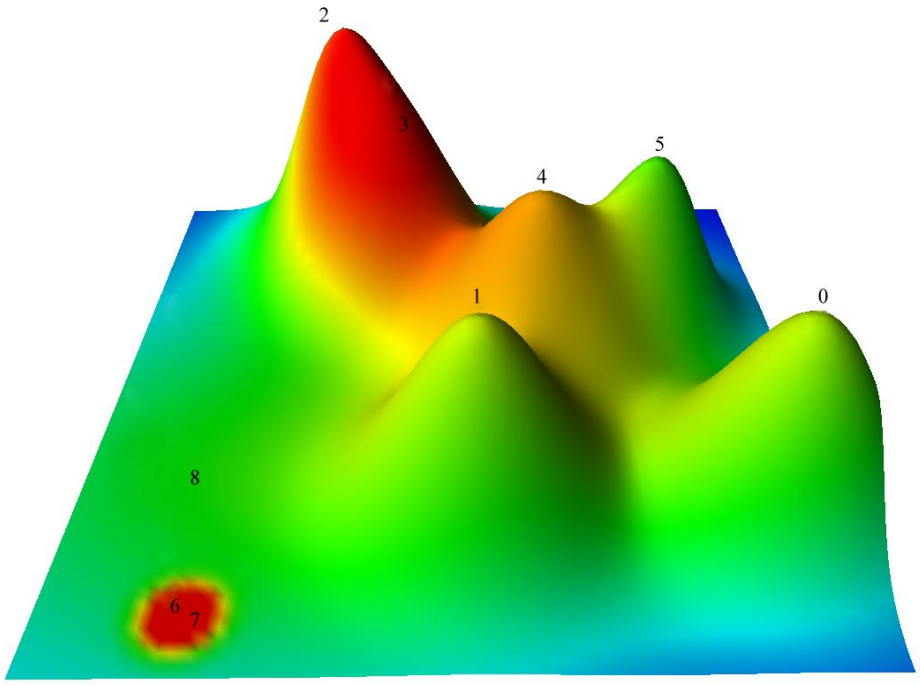

Рис. 19. График кластеров проанализированных изменений системы *Subversion*. Цифрами указаны соответствующие пикам номера кластеров

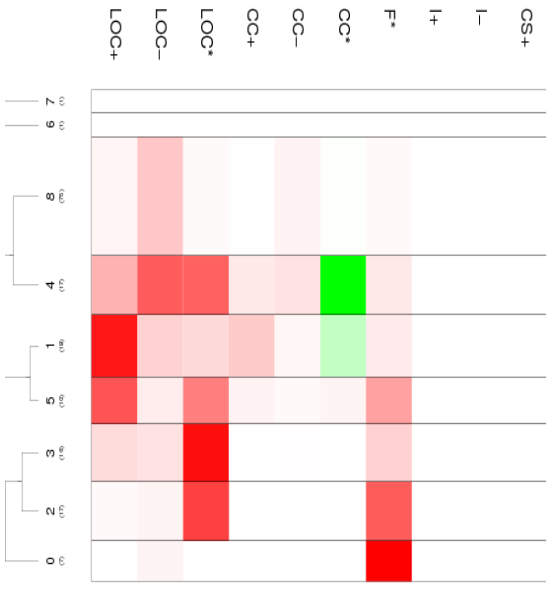

Рис. 20. Матрица кластеров, построенная для проанализированных изменений *Subversion*. По вертикали указаны номера кластеров, по горизонтали – названия метрик

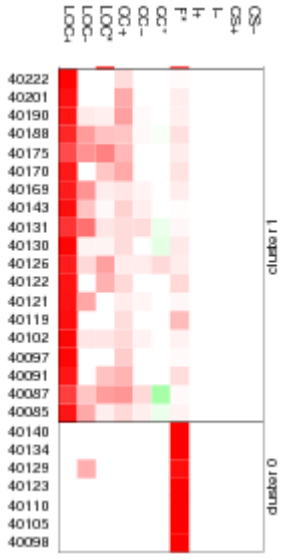

Рис. 21. Матрица изменений, построенная для кластеров 0 и 1 для проанализированных изменений *Subversion*. По вертикали слева указаны номера изменений (ревизии), справа – номера кластеров, по горизонтали – названия метрик

3.2.6. Общий эксперимент

Общий эксперимент по анализу изменений из нескольких программных систем проводится с целью установить среднее для всех проанализированных программных систем качество классификации, а также опытным путем обосновать гипотезу о возможности автоматизированной классификации.

Для проведения такого эксперимента были объединены проверочные множества изменений четырех из пяти проанализированных программных систем: *LRIT, e-Tutor, NHibernate, Subversion*. Входные данные эксперимента приведены в табл. 33.

Таблица 33. Входные данные эксперимента для систем *LRIT, e-Tutor, NHibernate, Subversion*

Общая информация	
Экспертные классы C	*Удаление кода, реализация новой функциональности, исправление логики, рефакторинг, форматирование кода с изменением комментариев*
Метод кластеризации метрик изменений	k-средних
Мера близости векторов метрик ρ	Косинус угла между векторами метрик
Набор метрик для кластеризации μ	Приведен в табл. 10
Входные данные метода	
Общее число классифицируемых изменений	203
Число размноженных изменений проверочного множества	41006
Данные настройки метода	
Выбранное число кластеров k	9, 10, 11, 12, 13

Эксперимент проведен с помощью метода складного ножа. Результаты эксперимента для различного числа кластеров приведены на графике в табл. 34.

Таблица 34. Результаты расчета предельных критериев качества кластеризации для проверки гипотезы о возможности автоматизированной классификации

Число кластеров	9	10	11	12	13
Значение P_{Qmin}	0,66	0,67	0,68	0,68	0,69
Значение E_{Qmax}	0,52	0,51	0,49	0,48	0,47

Наилучшим является результат, полученный для тринадцати кластеров, наихудшим – для девяти кластеров. Выберем результат для тринадцати кластеров, так как он наиболее оптимален по качеству разбиения и соответствует значениям, использованным в других экспериментах.

Для выбранных изменений проанализированных программных систем **подтверждается гипотеза о возможности автоматизированной классификации изменений методом кластеризации метрик** для тринадцати кластеров для следующих значений P_{Qmin} и E_{Qmax} с уровнем значимости 0,05:

$$P_{Qmin} = 0{,}69,\ E_{Qmax} = 0{,}47,$$

что свидетельствуют о приемлемом качестве автоматизированной классификации для практического применения метода, но при этом и о высокой степени неопределенности распределения изменений по кластерам. Следует заметить, что в общем эксперименте получен результат, который хуже, чем средний результат для частных экспериментов. Это объясняется тем, что в процессе кластеризации изменений каждой программной системы выделяются кластеры, специфичные для нее. При объединении изменений разных программных систем в рамках общего эксперимента происходит наблюдаемое снижение значений критериев качества.

3.2.7. Выводы по результатам экспериментов

Итак, результаты использования предложенного в диссертации метода для пяти программных систем следующие:

1. Экспериментально подтверждена возможность частичной автоматизации классификации изменений кода на основе кластеризации метрик. Выдвинутая во второй главе гипотеза о возможности автоматизированной классификации методом кластеризации метрик изменений подтверждается для значений P_Q не ниже 0,69 и E_Q не выше 0,47 на множестве из 203 изменений программных систем *NHibernate*, *LRIT, TEAS, Subversion.*
2. Экспериментально обосновано использование алгоритма *k-средних* с мерой близости объектов кластеризации (изменений), основанной на косинусе угла между векторами метрик для программных систем *NHibernate, LRIT, TEAS, Subversion*. Качество результата работы метода указанным алгоритмом с указанной мерой близости, позволяет использовать метод на практике.
3. Достигнуты следующие значения критериев качества классификации изменений: коэффициент Кохена согласованности автоматизированной и экспертной классификации κ = 0.79 для системы *Navi-Manager;* средняя чистота экспертных классов P_C = 0,75 ± 0,05, средняя энтропия экспертных классов E_C = 0,37 ± 0,06 для системы *NHibernate*; P_C = 0,71 ± 0,02, E_C = 0,43 ± 0,02 для системы *LRIT*, P_C = 0,73 ± 0,01, E_C = 0,46 ± 0,01 для системы *e-Tutor 5000*, P_C = 0,78 ± 0,01, E_C = 0,35 ± 0,02 для системы *Subversion*.
4. Применение предложенного метода позволяет существенно повысить степень автоматизации задачи классификации изменений. Для пяти программных систем участие эксперта требовалось в среднем для классификации одного из одиннадцати изменений, а остальные изменения классифицировались автоматически.

5. Предложенный метод позволяет повысить эффективность экспертизы изменений за счет отказа от рассмотрения от 30 до более чем 80 процентов изменений, не интересующих эксперта на текущей стадии разработки. Задача контроля внесения отдельных классов изменений упрощается за счет концентрации внимания эксперта на изменениях заданных классов.
6. Предложенный метод позволяет организовать обмен информацией об изменениях внутри команды разработки, который будет способствовать лучшему пониманию процесса участниками.
7. Предложенный метод позволяет проводить выборочную экспертизу изменений системы при ее анализе. Это помогает эффективно определять направление развития системы, происходившие модификации ее работы, а также степень контроля ее качества.
8. Предложенный метод позволяет улучшать качество кода посредством повышения эффективности его экспертизы, а также путем повышения информированности членов команды разработки о производимых в системе изменениях. Численный анализ степени повышения качества кода не проводился, в связи с принципиальной сложностью объективной его оценки. Эффективная экспертиза позволяет не только устранять алгоритмические ошибки, но и повышать читаемость кода, его расширяемость и многие другие важные характеристики качества.

В третьем разделе данной главы описано разработанное автором программное средство, реализующее предложенный в диссертации метод.

3.3. Описание программного инструмента автоматизированной классификации изменений

Разработанное автором диссертации программное средство *Code History Analyzer* состоит из следующих частей:

1. Модуль расчета метрик изменений, указанных в табл. 10 (модуль разработан автором, особенности расчета метрик изложены в разд. 2.2.2).
2. Модуль поиска редакционных предписаний (рассмотрены в разд. 2.2.2) изменений кода (основан на открытой системе контроля версий файлов *Subversion*).
3. Модуль кластеризации с оптимизацией функционала качества разбиения (основан на инструментальном средстве *CLUTO*, описание которого приведено в следующем разделе).

В четвертом разделе данной главы описаны особенности реализации алгоритма кластеризации в программном средстве *CLUTO*.

3.4. Особенности реализации используемого алгоритма кластеризации в программном средстве CLUTO

CLUTO – специализированное программное средство [61, 84], разработанное для кластеризации данных. Его особенностью является возможность задания параметра «число повторений» (*«ntrials»*), который указывает вычислить заданное число кластерных решений различными способами и выбрать лучшее из них на основе максимизации значения функционала качества кластеризации. Это позволяет улучшить качество кластеризации за счет снижения вероятности попадания значения функционала в локальный минимум. Кроме того, инструмент позволяет задавать различные алгоритмы для кластеризации, а также выбирать меру близости объектов кластеризации.

Выводы по главе 3

1. Применение предложенного метода позволяет существенно повысить степень автоматизации задачи классификации изменений. Для пяти

программных систем участие эксперта требовалось в среднем для классификации одного из одиннадцати изменений, а остальные изменения классифицировались автоматически при сохранении средних значений критериев качества классификации $P_C = 0{,}74 \pm 0{,}05$ (вероятность ошибки классификации произвольного изменения) $E_C = 0{,}40 \pm 0{,}06$ (степень неопределенности распределения изменений по классам). Также подтверждена гипотеза о возможности автоматизированной классификации методом кластеризации метрик изменений для значений критериев качества распределения изменений по кластерам P_Q не ниже 0,69 и E_Q не выше 0,47.

2. Количественный анализ повышения качества в результате применения метода не проводился. В диссертации ставилась цель повышения эффективности проведения экспертизы исходного кода в условиях ограниченности времени экспертов. Эта цель успешно достигается с помощью автоматизированной классификации изменений исходного кода и отбора изменений заданных классов для экспертизы. При этом наиболее типичными являются следующие классы изменений: *реализация новой функциональности, рефакторинг, исправление логики, удаление неиспользуемого кода, форматирование кода*.
3. Эффект от использования автоматизированной классификации следующий:
    - Задача контроля изменений, нежелательных на текущей стадии разработки, решается существенно более эффективно с помощью предложенного метода по сравнению с полной экспертизой изменений, так как позволяет выделять изменения классов, запрещенных на текущей стадии разработки с меньшими затратами времени.
    - Задача экспертизы изменений кода для изучения истории развития системы также выполняется существенно более эффективно.

4. В большинстве проектов по разработке программных систем, существует потребность по повышению эффективности экспертизы кода. Это может быть достигнуто с помощью оптимизации выбора изменений для экспертизы с помощью предложенного в работе метода автоматизированной классификации изменений. При этом для экспертизы выбираются изменения тех классов, которые наиболее важны на текущей стадии разработки.
5. Повышение эффективности экспертизы кода достигается за счет существенной экономии времени эксперта на классификации изменений, которая решается более эффективно с применением предложенного в диссертации метода.
6. Применимость метода для повышения эффективности экспертизы ограничена программными системами, при разработке которых существует ограничение времени на экспертизу исходного кода, не позволяющее выполнять ее полностью. Однако, исходя из опыта автора, большинство программных систем разрабатываются с ограничением во времени на экспертизу.
7. В таких условиях повышение качества разрабатываемой программной системы происходит за счет более эффективного проведения экспертизы кода благодаря концентрации эксперта на наиболее важных классах изменений кода. Разработанный метод автоматизированной классификации изменений позволяет выполнять классификацию изменений при минимальном участии эксперта.
8. Повышение эффективности экспертизы кода не только прямо способствует повышению качества кода за счет выбора наиболее важных изменений в условиях ограничения времени, но и позволяет использовать экспертизу для решения многих задач, связанных с обменом информации между участниками процесса разработки.

9. В процессе внедрения метода были выявлены следующие недостатки, которые предлагается устранять при разработке перспективных исследований по теме диссертации:
    - необходимость участия эксперта в процессе сопоставления кластеров классам;
    - отсутствие возможности классификации изменений, содержащих разнородные модификации.
10. Из изложенного следует, что применение метода на практике позволяет улучшить качество кода благодаря повышению эффективности процесса его экспертизы. Используя предлагаемый в работе подход, процесс экспертизы в условиях ограничения времени можно строить более эффективно с помощью отбора изменений наиболее важных классов изменений.

Заключение

В диссертации получены следующие результаты.

1. Обоснована возможность частичной автоматизации классификации изменений исходного кода методом кластеризации метрик.
2. Обоснован выбор метода k-средних с мерой близости объектов для кластеризации, основанной на косинусе угла между векторами метрик изменений.
3. Разработан метод автоматизированной классификации изменений исходного кода на основе кластеризации метрик изменений, позволяющий сократить число изменений для классификации, выполняемой вручную.
4. Разработано программное средство автоматизированной классификации изменений предложенным методом.

Перечисленные результаты получены в ходе выполнения совместных работ СПбГУ ИТМО и ЗАО «Транзас Технологии» и используются как при разработке программного обеспечения сложных систем, так и в учебном процессе.

В работе приводятся результаты сравнения классификаций изменений в программной системе с открытым исходным кодом, выполненные с использованием предложенного автоматизированного метода и вручную. Для этого привлекались эксперты, имеющие опыт разработки сложных программных систем не менее пяти лет. Всего в работе было проанализировано пять программных систем. Исследования показали, что метод позволяет сократить время классификации изменений по сравнению с «чисто ручной» экспертизой (для пяти программных систем участие эксперта требовалось в среднем для классификации одного из одиннадцати изменений, а остальные изменения классифицировались автоматически) при следующих средних значениях характеристик качества классификации: чистота $P_C = 0{,}74 \pm 0{,}05$ и энтропия $E_C = 0{,}40 \pm 0{,}06$. Также подтверждена гипотеза о возможности

автоматизированной классификации методом кластеризации метрик изменений для значений P_Q не ниже 0,69 и E_Q не выше 0,47.

В настоящее время в методе используется 11 метрик, перечень которых приведен в разд. 2.2. Автор предполагает, что возможно совершенствование качества классификации за счет увеличения числа используемых метрик, а также настройки на другие перечни классов изменений, кроме тех, которые рассмотрены в диссертации.

Из изложенного следует, что применение метода на практике позволяет улучшить качество кода благодаря повышению эффективности процесса его экспертизы. Используя предлагаемый в работе подход, процесс экспертизы в условиях ограничения времени можно строить более эффективно с помощью отбора изменений наиболее важных классов изменений.

Темы перспективных исследований

Перспективными являются следующие направления исследований:

1. Подбор метрик, позволяющих повысить качество работы предложенного метода за счет более четкого разделения изменений классов *рефакторинг* и *исправление логики*.
2. Разработка потокового метода классификации изменений на основе кластеризации метрик изменений;
3. Разработка модификации предложенного в диссертации метода на основе кластеризации метрик, основанной на нечеткой логике;
4. Применение предложенного в диссертации метода для оценки степени влияния изменений программного кода, не покрытых тестами на качество программной системы.

В указанных направлениях в настоящее время получены следующие результаты:

1. Сформулировано предположение, что метрики структуры кода (глубина дерева наследования, число наследуемых интерфейсов и другие), метрики связности классов (число классов, к которым происходит обращение из некоторого класса), а также метрики дублирования кода [2, 23, 24] позволят более четко выделять класс рефакторинг, что позволит повысить качество работы метода.
2. Разработан потоковый метод классификации изменений на основе кластеризации метрик изменений с использованием обучающего множества изменений. В настоящее время еще не исследован вопрос устойчивости алгоритма кластеризации метрик с использованием обучающего множества изменений. Также подробно не изучены особенности построения обучающего множества изменений для решаемой задачи [12].
3. Предложена модификация метода автоматизированной классификации изменений на основе нечеткой логики. Построена нечеткая

классификация множества изменений программной системы *Navi-Manager*. Проведен сравнительный анализ результата нечеткой классификации предложенным методом и экспертной классификации изменений [11].

4. Предложен теоретический способ автоматизированного контроля качества разработки на основе расчета метрики покрытия кода изменения тестами, с применением метода автоматизированной классификации изменений на два класса: изменение покрытого тестами кода и изменение непокрытого тестами кода [14].

Список литературы

1. *Ауэр К., Миллер Р.* Экстремальное программирование: постановка процесса. С первых шагов и до победного конца. СПб.: Питер, 2004, 368 с.
2. *Афанасьев С.В., Воробьев В.И.* Метрики для объектно-ориентированного проектирования сложных систем // Вестник гражданских инженеров, 2005, № 4, с. 118-123.
3. *Ахо А., Сети Р., Ульман Д. Д.* Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001, 768 с.
4. *Барсегян А. А., Куприянов М. С., Степаненко В. В., Холод И. И.* Методы и модели анализа данных: OLAP и Data Mining. СПб: БХВ-Петербург, 2004, 336 с.
5. *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. 375 с.
6. *Брукс П.* Метрики для управления ИТ-услугами. М.: Альпина Бизнес Букс, 2008, 288 с.
7. *Викиучебник*. Алгоритмы поиска редакционного предписания. http://ru.wikibooks.org/wiki/Редакционное_предписание
8. *Диаконис П., Эфрон Б.* Статистические методы с интенсивным использованием ЭВМ // В мире науки, 1993, №3, с. 60-72.
9. *Князев Е. Г.* Методы обнаружения закономерностей эволюции программного кода / Труды XIV Всероссийской научно-методической конференции «Телематика-2007». СПбГУ ИТМО. 2007. Т.2, с. 435–436.
10. *Князев Е. Г.* Применение автоматической классификации изменений программного кода для отслеживания реализации несанкционированной функциональности / V Санкт-Петербургская межрегиональная конференция «Информационная безопасность регионов России». СПб.: СПОИСУ. 2007, с. 84.

11. *Князев Е. Г.* Применение алгоритма нечеткой кластеризации метрик для классификации изменений программного кода / Сборник тезисов V Всероссийской межвузовской конференции молодых ученых. СПбГУ ИТМО. 2008, с. 282, 283.
12. *Князев Е. Г.* Применение алгоритма потоковой кластеризации для задачи классификации модификаций исходного кода / Тезисы докладов XV Международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». СПб ГПУ. 2008, с. 289, 290.
13. *Князев Е. Г. Шопырин Д. Г.* Использование автоматической классификации изменений программного кода в управлении процессом разработки программного обеспечения / Тезисы докладов Software Engineering Conference (Russia) (SECR–2007). М.: Tekama. 2007. Цифровой носитель.
14. *Князев Е. Г., Лобанов П. Г.* Оценка опасности изменений программного кода путем анализа покрытия кода модульными тестами / Сборник докладов X международной конференции по мягким вычислениям и измерениям. СПбГЭТУ «ЛЭТИ». 2007. Т.2, с. 273–275.
15. *Князев Е. Г., Шопырин Д. Г.* Автоматизированная классификация изменений программного кода методами многомерного статистического анализа // Информационные технологии. 2008. № 5, с. 48–53.
16. *Князев Е. Г., Шопырин Д. Г.* Анализ изменений программного кода методом кластеризации метрик // Научно-технический вестник СПбГУ ИТМО. Исследования в области информационных технологий. 2007. Вып. 39, с. 197–208.
17. *Князев Е. Г., Шопырин Д. Г.* Анализ изменений программного кода методом кластеризации метрик / Сборник тезисов IV межвузовской конференции молодых ученых. СПбГУ ИТМО. 2007, с. 68.
18. *Князев Е. Г., Шопырин Д. Г.* Использование автоматизированной классификации изменений программного кода в управлении процессом

разработки программного обеспечения // Информационно-управляющие системы. 2008. № 5, с. 15–21.

19. Кобзарь А. Прикладная математическая статистика. Для инженеров и научных работников. М.: Физматилит, 2006. 816 с.
20. *Количественные методы в исторических исследованиях* / Под ред. Ковальченко И. Д. М.: Высшая школа, 1984. 384с.
21. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов. Докл. АН СССР, 163, 4, 1965, с. 845-848.
22. *Липаев В. В.* Методы обеспечения качества крупномасштабных программных средств. М.: Синтег, 2003, 520 с.
23. *Липаев В.В.* Выбор и оценивание характеристик качества программных средств: Методы и стандарты. М.: Синтег, 2001. 224 с.
24. *Макконелл С.* Совершенный код. СПб: Питер, 2007, 896 с.
25. *Мандель И.Д.* Кластерный анализ. М.: Финансы и статистика, 1988. 176 с.
26. *Мартин Р.* Быстрая разработка программ: принципы, примеры, практика. М.: Вильямс, 2003, 752 с.
27. *Новиков Ф. А.* Дискретная математика для программистов. СПб.: Питер, 2003, 460 с.
28. Орлов А. И. О реальных возможностях бутстрепа как статистического метода // Заводская лаборатория, 1987, Т.53, № 10, с. 82-85.
29. *Орлов С. А.* Технологии разработки программного обеспечения: Учебник для вузов. СПб: Питер, 2004. 528 с.
30. Подборка статей по бутстрепу // Заводская лаборатория, 1987, Т.53, № 10, с. 76-99.
31. *Чубукова И.А.* Data Mining. М.: Лаборатория Базовых Знаний, 2008. 382 с.
32. *Фаулер М.* Рефакторинг: улучшение существующего кода. СПб: Символ-Плюс, 2003. 432 с.
33. *Ханк Д. Э., Уичерн Д. У., Райтс А. Д.* Бизнес-прогнозирование. 7-е издание, М.: Вильямс, 2003, 656 с.

34. *Эфрон Б.* Нетрадиционные методы многомерного статистического анализа. М.: Финансы и статистика, 1988, 263 с.
35. *A Guide to the Project Management Body of Knowledge* (PMBOK® Guide) – Fourth Edition. Project Management Institute, USA, 2008, 459 p.
36. *Aoyama M.* Agile software process and its experience / Proceedings of the 20th international Conference on Software Engineering (Kyoto, Japan, April 19–25). International Conference on Software Engineering. 1998. IEEE Computer Society, Washington, DC, pp. 3–12.
37. *Beck K.* Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, USA, 2004, 224 p.
38. *Beck K.* Test driven development by example. Addison-Wesley Professional, USA, 2002, 240 p.
39. *Bevan J., Whitehead E. J., Kim S., Godfrey M.* Facilitating Software Evolution with Kenyon / European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, 2005, pp. 177–186.
40. *Canfora G., Cerulo L., Penta D. M.* Identifying Changed Source Code Lines from Version Repositories / MSR: International Workshop on Mining Software Repositories. Minneapolis, USA. 2007, pp. 34–45.
41. *Chidamber S. R., Kemerer C. F.* A Metrics Suite for Object Oriented Design // IEEE Transactions on Software Engineering, Vol. 20(6), June 1994, pp. 62–69.
42. *Cohen J.* A Coefficient of Agreement for Nominal Scales // Educational and Psychological Measurement. 1960, № 2, pp. 37–46.
43. *Cohen J.* Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice). http://smartbearsoftware.com, USA, 2006, 233 p.
44. *Collard M. L.* Meta-Differencing: An Infrastructure for Source Code Difference Analysis. Kent State University, Kent, Ohio USA. Ph.D. Dissertation Thesis, 2004. 136 p.
45. *Collins-Sussman B., Fitzpatrick B.W., Pilato M.* Version Control with Subversion. O'Relly, 2004. http://svnbook.red-bean.com/

46. *Concurrent Versions System* (CVS). http://www.nongnu.org/cvs/
47. *Crane Simulator*. Transas. http://transas.com/products/default.asp
48. *Demeyer S., Ducasse S., Nierstrasz O.* Finding refactorings via change metrics / Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00), pp. 166–178, 2000.
49. *Efron B.* Bootstrap methods: Another look at the jackknife / Ann. Statist., № 1, 1979, pp. 1-26.
50. *Emam E. K.* Benchmarking Kappa for Software Process Assessment Reliability Studies. Technical Report ISERN-98-02, International Software Engineering Research Network, 1998. http://citeseer.ist.psu.edu/elemam98benchmarking.html
51. *Engine Room Simulator (ERS) 4000*. Transas. http://www.transas.com/products/simulators/sim_products/ers/
52. *E-Tutor 5000.* Transas. http://transas.com/products/default.asp
53. *Fagan, M. E.* Design and Code Inspections to Reduce Errors in Program Development // IBM Systems Journal, № 3, 1976, pp. 182–211.
54. *Free Software Foundation. GNU Compiler Collection.* http://www.gnu.org
55. *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
56. *HyppoDraw*. http://www.slac.stanford.edu/grp/ek/hippodraw/index.html
57. *IBM Object-Oriented Technology Center*. Developing Object-Oriented Software: An Experience-Based Approach. Prentice Hall, IBM, 1997, 636 p.
58. *Joachims T.* Text Categorization with Support Vector Machines: Learning with Many Relevant Features / ECML 98, 10th European Conference on Machine Learning, Chemnitz, Germany, 1998, pp. 137–142.
59. *Kagdi H., Collard M., Maletic J.* Towards a Taxonomy of Approaches for Mining of Source Code Repositories / Proceedings of the 2005 international workshop on Mining software repositories MSR '05. ACM SIGSOFT Software Engineering Notes. St. Louis, Missouri, 2005, pp. 1–5.

60. *Kaner C.* Exploratory Testing, Florida Institute of Technology / Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, USA, 2006, pp. 36–39.
61. *Karypis G.* CLUTO. A Clustering Toolkit. Technical Report: #02-017, University of Minnesota, Department of Computer Science Minneapolis, USA, 2003, 71 p.
62. *KDE*. K Desktop Environment. http://www.kde.org/
63. *Kim M., Notkin D.* Program element matching for multi-version program analyses / MSR ’06: Proceedings of the 2006 International Workshop on Mining Software Repositories. 2006, pp. 58–64.
64. *Kim S., Whitehead E. J., Zhang Y.* Classifying Software Changes: Clean or Buggy? http://www.cs.ucsc.edu/~ejw/papers/cc.pdf
65. *Knyazev E.* Automated Source Code Changes Classification for Effective Code Review and Analysis / Proceedings of SYRCoSE 2008 (SYRCoSE) Spring Young Researchers Colloquium on Software Engineering. Volume 2, pp. 55–59.
66. *Kruskal J. B., Wish M.* Multidimensional scaling. // Paper Series on Quantitative Applications in the Social Sciences, (№№ 07–011), 1978.
67. *Lamping J., Rao R., Pirolli P*. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies // Proc. ACM Conf. Human Factors in Computing Systems, CHI, ACM, 1995, pp. 401–408.
68. *Larman C., Basili V. R.* Iterative and Incremental Development: A Brief History // Computer, № 6, 2003, pp. 47-56
69. *Liquid Cargo Handling Simulator (LCHS)*. Transas. http://www.transas.com/products/simulators/sim_products/ers/
70. *Long range identification and tracking (LRIT)*. International Maritime Organization. http://www.imo.org/safety/mainframe.asp?topic_id=905
71. *Lorenz M., Kidd J.*, Object-Oriented Software Metrics: A Practical Approach, PrenticeHall, 1994, 154 p.

72. *Maitra R.* Clustering Massive Datasets With Application in Software Metrics and Tomography // Technometrics. № 3, 2001, pp. 336–346.
73. *Maletic J. I., Collard M. L.* Supporting Source Code Difference Analysis / Proceedings of IEEE International Conference on Software Maintenance (ICSM'04). Chicago, Illinois. September 11–17, 2004, pp. 210–219.
74. *McCabe T. J.* A Complexity Measure // IEEE Trans SE-2 № 4, December 1976.
75. *Mockus A., Votta L. G.* Identifying reasons for software change using historic databases / Proceedings of the International Conference on Software Maintenance (ICSM). San Jose, California, 2000, pp. 120–130.
76. *MSqared Resource Standard Metrics Documentation.* http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm
77. *MSqared Resource Standard Metrics.* Code Analysis for Code Reviews and Acceptance. http://msquaredtechnologies.com/m2rsm/index.htm
78. *Navi-Manager Vessel Monitoring System.* Transas. http://www.transas.com/products/shorebased/manager/ http://www.transas.ru/products/shorebased/fleet/navi-manager/
79. *Navi-Trainer Professional (NTPro) 5000*. Transas. http://www.transas.com/products/simulators/sim_products/navigational/
80. *NHibernate*. Object-relation mapping for Microsoft .NET. https://www.hibernate.org/343.html
81. *Pan K., Kim S., Whitehead J. E.* Bug Classification Using Program Slicing Metrics / Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), Philadelphia, PA, 2006, pp. 77–86.
82. *Radice R.* Software Inspections. Software Technology Transition, www.stt.com // Methods and Tools, № 3, Martinig & Associates, USA, 2009, pp. 7–20.
83. *Raghavan S., Rohana R., Podgurski A., Augustine V.* Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases / Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04). Chicago, Illinois, September 11–14, 2004, pp. 188–197.

84. *Rasmussen M.* gcluto: A graphical interface for clustering algorithms and visualizations. Technical Report TR# 04–020, Department of Computer Science & Engineering, University of Minnesota, Minneapolis, USA, 2004, 10 p.
85. *Robbes R.* Mining a Change-Based Software Repository / MSR: International Workshop on Mining Software Repositories. Minneapolis, USA, 2007, pp. 120–124.
86. *Scott S., Matwin S.* Feature Engineering for Text Classification / Sixteenth International Conference on Machine Learning, Bled, Slovenia, 1999, pp. 379–388.
87. *Sebastiani F.* Machine Learning in Automated Text Categorization // ACM Computing Surveys, № 1, 2002, pp. 1–47.
88. *Sliwerski J., Zimmermann T., Zeller A.* When Do Changes Induce Fixes? / Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, 2005, pp. 24–28.
89. *Smart Bear Sofware.* http://smartbear.com/
90. Software release lifecycle. http://en.wikipedia.org/wiki/Software_release_life_cycle
91. *Subversion* (SVN). http://subversion.tigris.org
92. *The Apache Software Foundation.* Apache HTTP Server. http://httpd.apache.org.
93. *TortoiseSVN*. Windows Shell Extension for Subversion. http://tortoisesvn.tigris.org/
94. *Zimmermann T.* Knowledge Collaboration by Mining Software Repositories / Proceedings of the 2nd International Workshop on Supporting Knowledge Collaboration in Software Development (KCSD 2006). Tokyo, Japan, 2006, pp. 64, 65.
95. *Zimmermann T., Wei.gerber P., Diehl S., Zeller A.* Mining Version Histories to Guide Software Changes / Proceedings of 26th International Conference on Software Engineering (ICSE'04). Edinburgh, Scotland, United Kingdom, May 23–28, 2004, pp. 563–572.