УДК 004.415.53

ИСПОЛЬЗОВАНИЕ СРЕДСТВ СТАТИЧЕСКОЙ ОТЛАДКИ ДЛЯ ВЕРИФИКАЦИИ ПРОГРАММНОЙ СРЕДЫ SCOPESHELL

С. В. Степанов[1], **А. Г. Шишкин**[1]

Рассматриваются свободно распространяемые пакеты для статической верификации кодов, написанных на языке Java. Приведены результаты тестирования и оценка эффективности применения различных верификаторов на примере программной системы ScopeShell, разработанной на кафедре автоматизации научных исследований факультета ВМиК МГУ.

Ключевые слова: статическая верификация, отладка программного обеспечения, автоматизированная отладка, язык программирования Java.

1. Введение. С каждым годом растет функциональность, а вместе с этим увеличивается и средний размер компьютерных кодов. Как известно, при разработке больших программных продуктов возникает проблема их последующей верификации и валидации. Следует разграничивать понятия "верификация" и "валидация". Несмотря на широко распространенное мнение об их идентичности, между ними существует значительное различие. Верификация — проверка соответствия разрабатываемого продукта ранее установленной спецификации (низкоуровневая проверка). При верификации проверяется, прежде всего, правильность создания программного обеспечения (ПО). Валидация — проверка правильности выполнения разработанным ПО поставленной задачи, т.е. полученный программный продукт должен корректно выполнять именно те функции, которые требуются конечному пользователю.

Следует отметить, что к верификации относится статическая отладка (вид отладки, при которой сам программный продукт как таковой не задействован, проверяется исходный код, синтаксис и семантика, наличие ошибок), а к валидации — динамическая отладка (отладка, при которой требуется запуск кода, проверка логики и функциональности при различных условиях).

До недавнего времени верификация программного обеспечения проводилась вручную, что требовало больших временны́х затрат и следствием чего являлся пропуск многих ошибок. Однако в последнее время для статической отладки разрабатывается большое число различных средств, называемых статическими анализаторами или верификаторами, призванных облегчить процесс поиска ошибок и их исправления в программе. Статические анализаторы проверяют исходный текст программы и выдают сообщения о подозрительных строках кода, которые могут оказаться ошибочными. Совершенно очевидно, что все хорошие анализаторы должны обладать достаточно высокой точностью распознавания. Это означает, что, с одной стороны, анализаторы не должны пропускать подозрительные части кода, а с другой — не должны выдавать слишком много ложных сообщений.

Так как в данной работе рассматривается автоматическая верификация ПО, написанного на языке Java, то, естественно, все приведенные далее средства отладки относятся именно к Java-приложениям. Нами были рассмотрены свободно распространяемые статические анализаторы языка Java, начиная от простейших и заканчивая наиболее совершенными, требующими для получения адекватных результатов добавления в исходный код комментариев на языке JML. Работа верификаторов была проанализирована на большой программной системе ScopeShell, описание которой дано ниже.

2. Программная среда ScopeShell. Традиционно сложные численные коды для решения многих физических задач разрабатываются без специального графического интерфейса, что повышает их мобильность, под которой понимается свойство, позволяющее выполнять программу на различных компьютерах с минимальными изменениями или без них. Такой подход объясняется необходимостью использования разнообразных мощных удаленных компьютеров и постоянным, достаточно быстрым, совершенствованием вычислительной техники и системного программного обеспечения. Сильная привязка ресурсоемкого численного кода к конкретному графическому пакету является плохим стилем, так как в итоге сужает круг применимых вычислительных систем.

Настройка кода на конкретный расчет, компиляция и запуск программы на выполнение, как правило, проводятся вручную. Каждому пользователю кода необходимо помнить все имена файлов, содержащих

[1] Московский государственный университет им. М. В. Ломоносова, факультет вычислительной математики и кибернетики, Ленинские горы, 119992, Москва; С. В. Степанов, студент, e-mail: sergey.v.stepanov@gmail.com; А. Г. Шишкин, ст. науч. сотр., e-mail: shishkin@cs.msu.su

входную информацию (это могут быть не только файлы с данными, но и с программами), знать особенности компиляции и запуска кода на том или ином компьютере. Задача осложняется еще и тем, что в большинстве случаев вычисления требуют значительных ресурсов ЭВМ. Вследствие этого необходимо задействовать крупные вычислительные комплексы, к которым обычно имеется только удаленный доступ по строго определенным защищенным протоколам связи (SSH, SFTP и др.).

Визуализация данных вызывает еще бóльшие трудности. Развитые численные коды генерируют сотни и тысячи файлов с данными. Для построения графиков необходимо помнить смысл содержимого файлов, используемые единицы измерения величин, формат данных, создавать подписи к рисункам, осям координат, кривым и т.п.

Обычно применяют универсальные графические пакеты общего назначения, многие из которых требуют специального, не всегда простого, формата входных данных, задания множества настроек или даже написания программы на некотором внутреннем языке. Такие системы хороши для качественного построения единичных графиков, но малопригодны для мониторинга вычислений, особенно удаленных, и быстрой визуализации большого множества функций.

Система ScopeShell, представляющая собой универсальный дружественный графический интерфейс для численных кодов и визуализации данных, облегчает решение указанных проблем. Она предназначена для автоматизации рутинных операций настройки входных и выходных данных численного кода, его компиляции и запуска, мониторинга формирования данных, преобразования формата данных, построения двумерных и трехмерных графиков.

Одно из преимуществ ScopeShell — платформенная независимость графического интерфейса, который реализован на языке Java.

Другим достоинством является использование различных графических пакетов для непосредственного изображения данных, что позволяет избежать реализации сложной задачи визуализации и сосредоточиться на интерфейсе пользователя. ScopeShell совместим с пакетом Gnuplot [1], разрабатываемым с 1986 года, свободно распространяемым и доступным в версиях для различных операционных систем, в том числе, MS Windows и UNIX.

Однако ScopeShell не привязан к конкретному графическому пакету. Общение с Gnuplot построено с помощью формирования файла директив, содержание которого может быть легко адаптировано для другого графического пакета.

Работа со ScopeShell состоит в манипуляции с древовидными списками и таблицами. Настройка ScopeShell на конкретный численный код проводится один раз и сохраняется в виде проекта. Проекты можно копировать и модифицировать, что позволяет избежать введения большого объема информации заново. Имеется рабочее поле для сохранения определенных, например часто используемых, графиков. Процедура создания сложного проекта в ScopeShell может занять некоторое время. Однако работа с готовым проектом проходит быстро и удобно, фактически с помощью нескольких щелчков мыши.

Операции над полями таблиц унифицированы. Детали описаны в он-лайн документации, реализованной в виде Web-страниц. Действия пользователя аналогичны принятым, например, в известной системе MS Excel.

К достоинствам ScopeShell относится наличие гибкой контекстно-зависимой системы логических справок ScopeShell Wizard, автоматически подсказывающей возможные варианты действий в конкретной ситуации и предоставляющей информацию по каждому варианту.

Для хранения результатов вычислительного эксперимента и информации о соответствующих файлах в рамках системы ScopeShell разработана подсистема, состоящая из трех основных компонентов: сервера баз данных, программы-демона на стороне сервера баз данных и клиентской части. В качестве сервера баз данных используется MySQL Server, являющийся свободно распространяемым в версиях как для Windows, так и для UNIX.

Вся хранящаяся информация разбита на четыре уровня. Первым уровнем является информация о проекте: название проекта, данные о руководителе, даты начала и последнего обновления, количество задач и комментарии к проекту. На втором уровне находится информация о задаче, т.е. информация о том, кто является ответственным за данную задачу, какова цель в проекте, о дате начала задачи, о количестве экспериментов. Кроме того, пользователь может хранить некоторые комментарии, касающиеся данной задачи. Третьим уровнем являются сведения о результатах вычислительного эксперимента, которые представляют собой набор масок для выделения необходимых данных, файлы масок и сведения о файлах данных. И, наконец, последним уровнем являются сами данные вычислительного эксперимента.

Система поддерживает контекстный поиск элементов, импорт новых файлов данных, ввод новых проектов, задач, масок, изменения данных проектов, задач, файлов, масок, экспорт файлов данных из

базы данных, экспорт проектов программной системы ScopeShell из базы данных, импорт проектов программной системы ScopeShell в базу данных, преобразование данных в различные форматы, например NetCDF.

Систему можно использовать для доступа как к локальной базе данных, так и к удаленной. Удаленный доступ может обеспечиваться по протоколу JDBC и через безопасное соединение по протоколу SSH. Для этого в системе реализован модуль доступа SSHClient, который позволяет не только перенаправлять запросы программной системы к базе данных по протоколу SSH, но и пересылать файлы по протоколу SFTP.

3. Программные средства для статического анализа. В настоящее время существует довольно много свободно распространяемых верификаторов Java-приложений. В первую очередь такие средства отличаются по способу анализа кода — большинство программ анализируют исходный код, но некоторые могут анализировать и байт-код. Далее приведены основные характеристики статических верификаторов, которые были использованы нами для отладки программной системы ScopeShell. В то время как некоторые из них являются совсем простыми, другие предполагают использование специальных языков, например JML.

3.1. FindBugs. Пакет, разработанный в университете Мэриленда (США), является простым в обращении и в то же время очень функциональным [2, 3]. FindBugs имеет простой и интуитивно понятный графический интерфейс, при этом сохранена возможность запуска через командную строку, а также использование пакета как задания для сборщика проектов Ant (рис. 1). Для FindBugs также имеются плагины для различных интегрированных сред разработки программного обеспечения (IDE), в том числе наиболее популярной среды Eclipse.

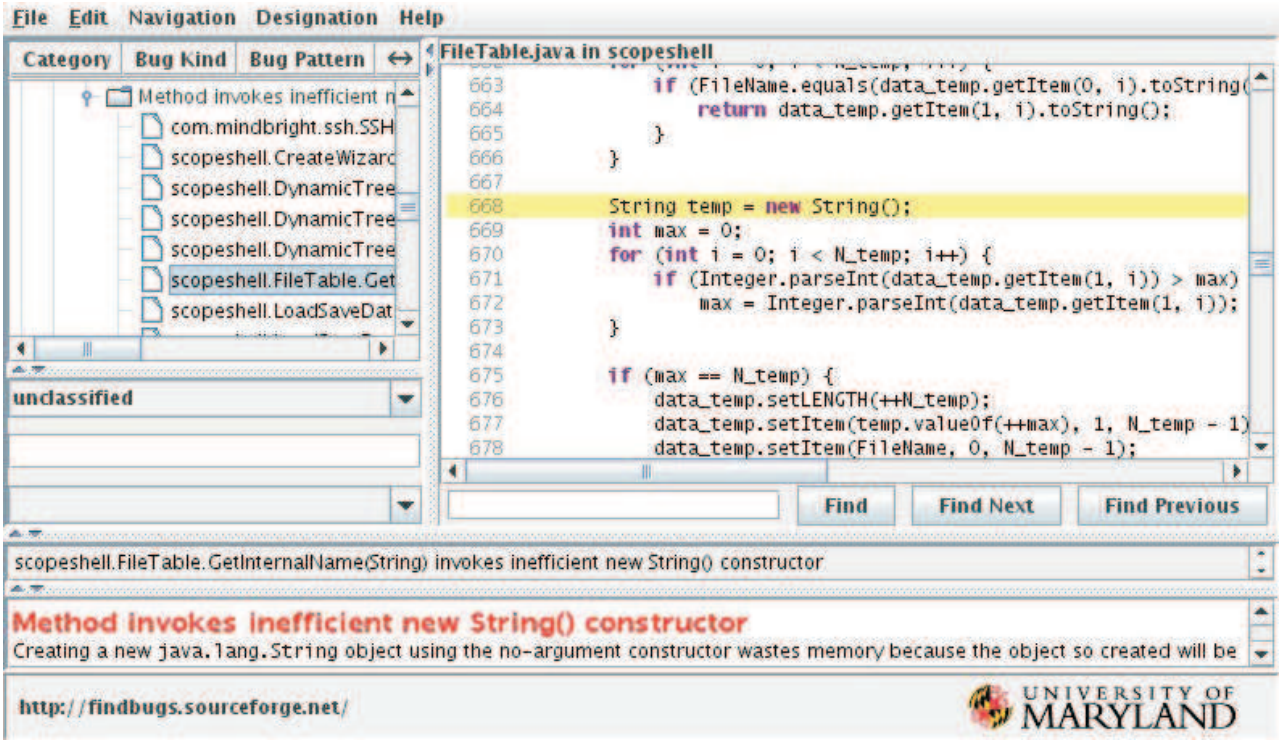

Рис. 1. Скриншот пакета FindBugs

Пакет анализирует байт-код с помощью BCEL (Byte Code Engineering Library) [4]. Помимо этого, существует возможность кроме байт-кода исследовать также исходный код, тогда при анализе найденных ошибок появится возможность просмотра проблемных мест в исходном коде.

Результаты работы пакета выводятся в виде дерева, ошибки и предупреждения разделены по классам, можно делать комментарии и присваивать статус ошибкам, что значительно упрощает анализ результатов. Программа находит как незначительные недочеты, так и серьезные ошибки. На экран можно вывести отчет в виде таблицы, а также сделать экспорт результатов в xml формат.

3.2. Pmd. В отличие от FindBugs, пакет pmd работает с исходным кодом Java [5]. У пакета нет графической среды, программу нужно запускать в командной строке (рис. 2). Для начинающего разработчика

это может показаться большим минусом на фоне развитого интерфейса FindBugs, но на самом деле для работы с пакетом pmd достаточно использовать несколько команд. Программа находит не только ошибки в коде, но и так называемый dead-code (код, который нигде не используется в программе), что не является ошибкой, но в то же время приводит к увеличению объема исходного кода и усложнению его анализа для разработчика. Pmd способен обнаружить такие ошибки, как неиспользуемые локальные переменные, пустые условные блоки if() и конструкции try/catch/finally/switch, неиспользуемые private-методы в классе и другие. Одним из достоинств pmd является возможность использования различных правил анализа, в том числе созданных самим пользователем.

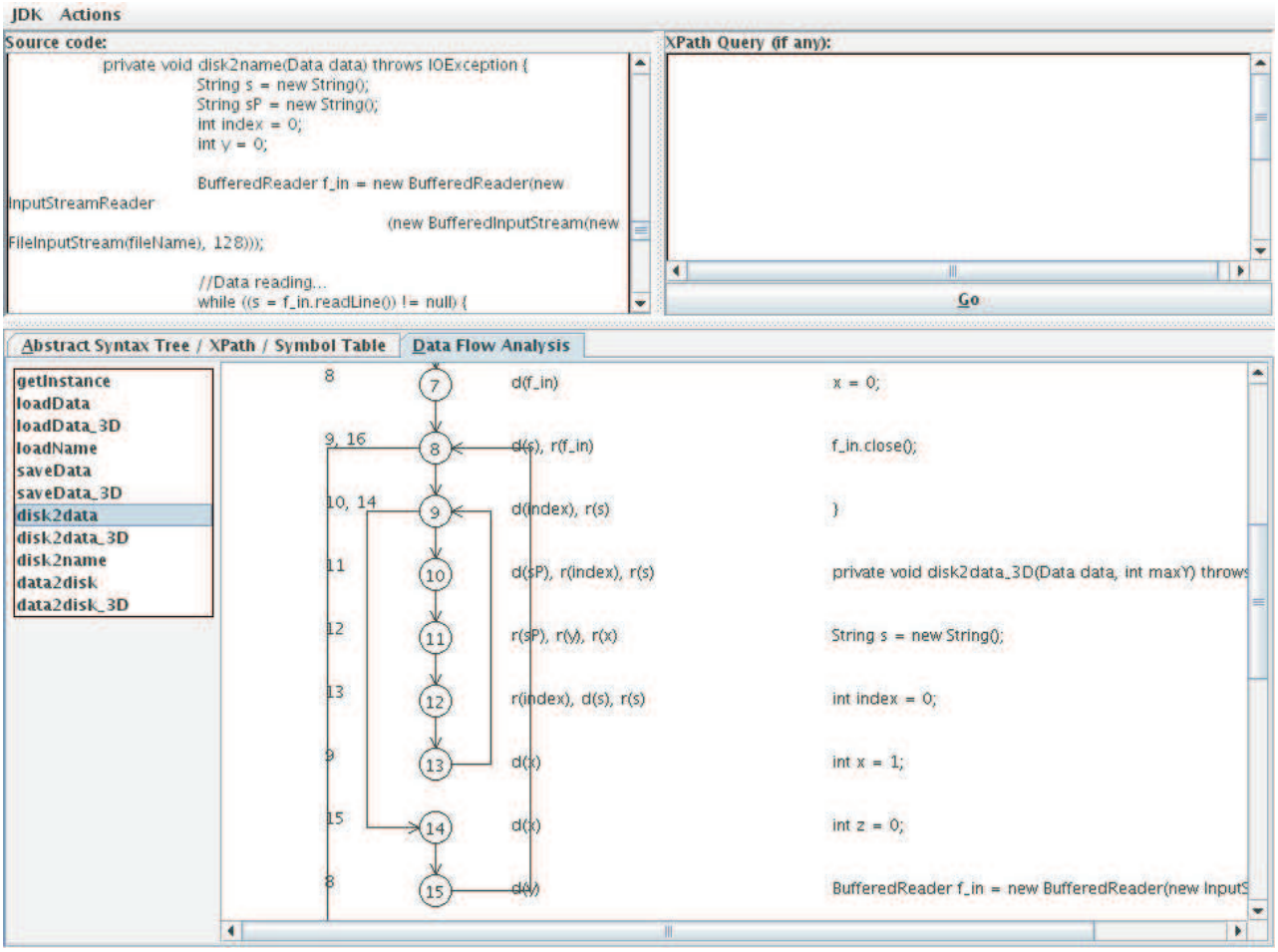

Рис. 2. Скриншот пакета pmd

Отметим, что на сайте данного верификатора представлено большое количество плагинов практически для всех интегрированных сред разработки IDE, что, несомненно, является большим плюсом. Кроме того, разработчики предоставили плагины для сборщиков проектов Ant и Maven, что очень удобно, особенно при анализе больших проектов.

Помимо самого анализатора, в пакет pmd входит отдельная утилита с графической оболочкой для поиска повторяющегося кода и программа для создания правил анализа со встроенным анализатором структуры кода.

3.3. Lint4j. Lint4j (Lint for Java) — интересный пакет для статического анализа как исходного, так и байт-кода программы [6]. Графический интерфейс, как и в pmd, отсутствует, но запустить программу можно довольно легко через командную строку. Программа способна находить различные ошибки, связанные с проблемами синхронизации и со взаимной блокировкой (deadlock) потоков, быстродействием и ресурсоемкостью, распределением памяти, а также использованием платформозависимых конструкций (например, вызовов System.exec()). Как и pmd, lint4j имеет плагины для различных сред разработки, а также для сборщиков проектов Ant и Maven.

3.4. JLint. Пакет, написанный на языке C++, состоит из двух модулей — AntiC (синтаксический анализ кода на C, C++, Java) и JLint (семантический анализ, поиск deadlocks в многопоточных про-

граммах) [7, 8]. Для работы с программой требуется предварительно ее скомпилировать. JLint позволяет получать три разных типа сообщений, относящихся к синхронизации, наследованию и потокам данных. При этом каждый тип сообщений в свою очередь подразделяется на несколько категорий, включающих по крайней мере одно сообщение. Данный пакет позволяет легко включать и выключать как отдельные сообщения, так и целые их категории. Основным достоинством Jlint является высокая скорость обработки данных.

3.5. ESC/Java. Данный пакет, разрабатываемый группой KindSoftware с 2003 г. и являющийся анализатором исходного кода [9], поддерживает язык JML (Java Modeling Language) [10, 11]. В пакете реализован графический интерфейс, однако можно работать и через командную строку (по замечаниям разработчиков, этот способ является основным).

Отметим: для запуска пакета предварительно нужно задать две переменные окружения (SIMPLIFY и ESCTOOLS_ROOT). Для работы через интерфейс достаточно создать новый проект и загрузить исходный код программы для анализа. Результат выводится в виде дерева, поэтому обрабатывать информацию довольно легко (рис. 3).

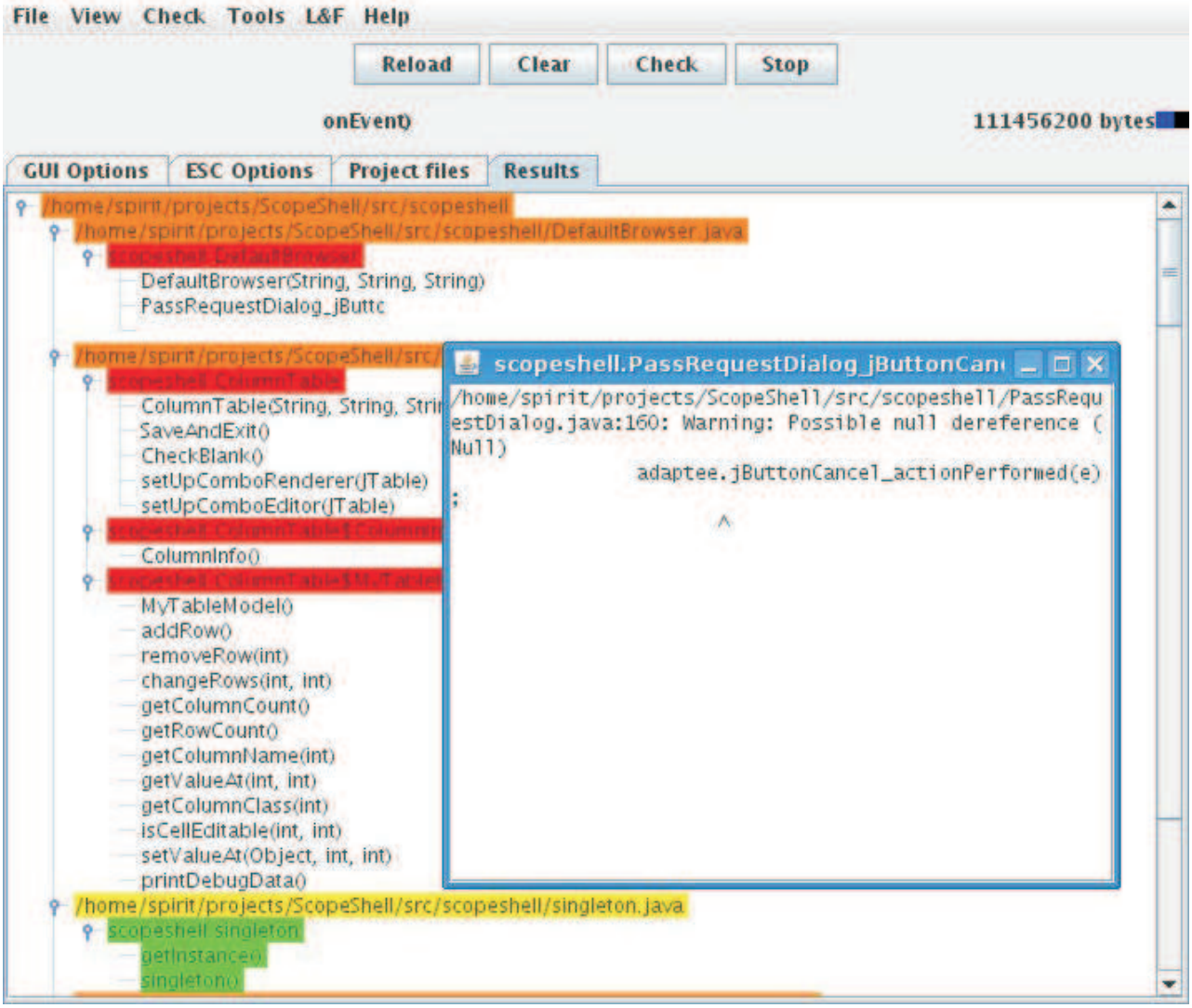

Рис. 3. Скриншот пакета ESC/Java

Программа поддерживает код, написанный на Java версии 1.4 и ниже. Заметим, что на сайте разработчиков сказано, что пакет может некорректно работать на JVM версии выше 1.5, однако в нашем случае программа исправно работала в среде Linux с использованием JVM версии 1.6. С помощью JML в исходном коде можно сделать комментарии, предотвратив тем самым многие предупреждения при анализе. Мы не будем углубляться в описание механизма работы программы с комментариями JML, отметим лишь, что с помощью комментариев на языке JML, называемых прагмами (от англ. “pragma”), можно зна-

чительно повысить эффективность отслеживания ошибок в коде. Оптимальным вариантом будет запуск анализа без использования прагм, затем, учитывая результаты анализа, внесение необходимых комментариев и повторный запуск пакета. Полная документация по использованию языка JML доступна на сайте разработчика, а также в пакет включено множество примеров по использованию JML для статической верификации исходного кода.

3.6. Hammurapi. Пакет Hammurapi включает в себя модули Mesopotamia и Hammurapi Rules, каждый из них может быть изменен и оптимизирован под конкретный проект [12]. Особенностью Hammurapi является его способ анализа кода и возможность работать не только с кодом на языке Java, но и с другими языками и форматами (например, xml-формат данных или JavaScript и AJAX).

Для запуска пакета Hammurapi необходимо сделать несколько шагов. Первый из них заключается в запуске базы данных HSQLDB, поставляемой в пакете Mesopotamia, и инициализации java-модуля для Mesopotamia. Для контроля содержимого БД можно запустить менеджер HSQLDB. Следующим шагом является загрузка данных в БД. Для этого потребуется создание класса-загрузчика.

Когда данные уже загружены в БД, можно непосредственно запустить пакет Hammurapi, указав идентификационный номер репозитория, в который были загружены данные в БД. Просмотр результатов анализа возможен через web-интерфейс, для запуска которого потребуется servlet-контейнер (например, Apache Tomcat, который уже включен в пакет Hammurapi).

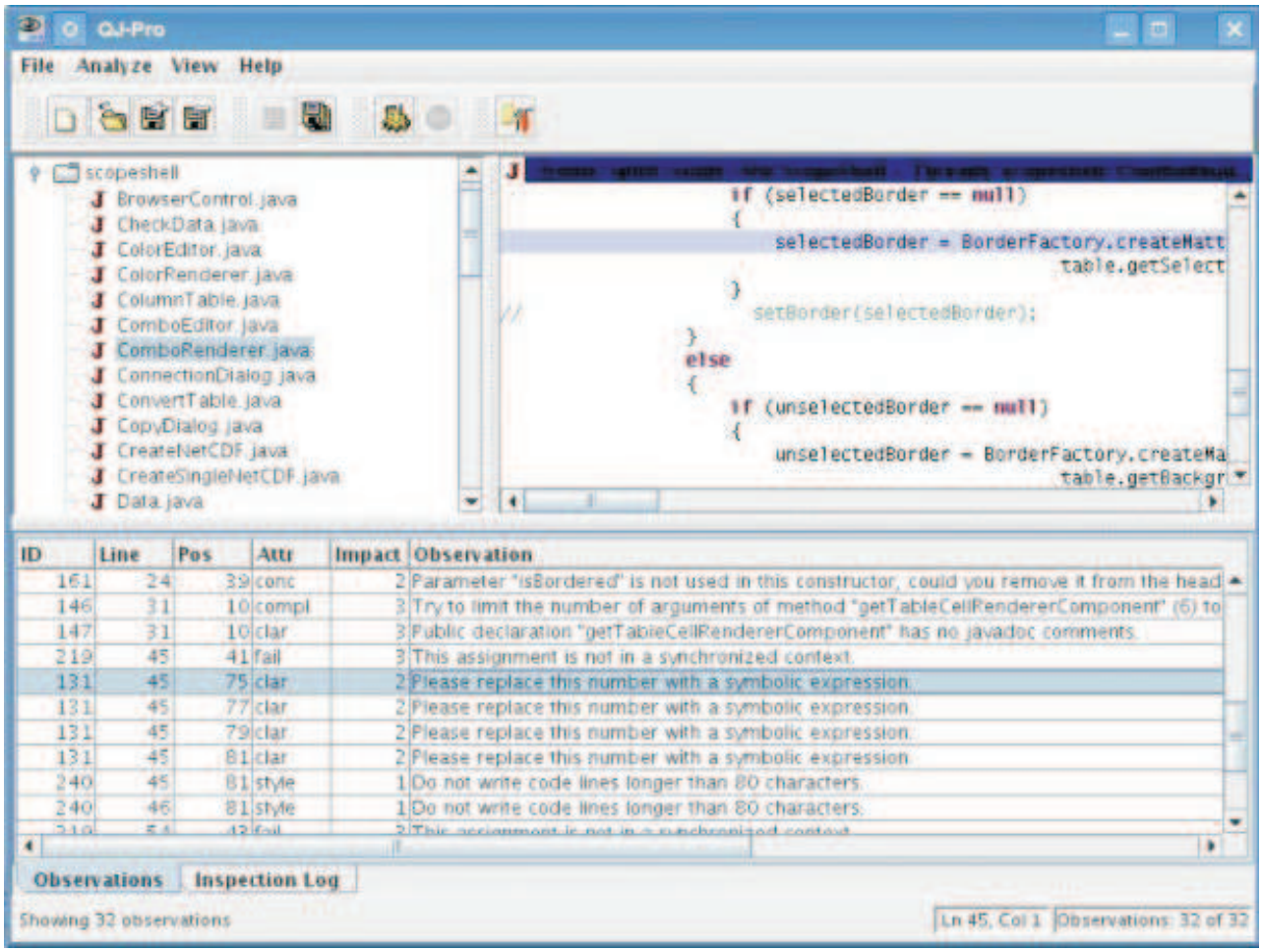

Рис. 4. Скриншот пакета QJ-Pro

Одним из достоинств пакета Hammurapi является поддержка различных правил анализа. Правила анализа Hammurapi Rules также разрабатываются на Java, поэтому процесс создания новых правил не создаст проблем разработчику.

Работа с пакетом не является тривиальной, однако результатом некоторых усилий станет подробный анализ исходного кода. Отметим, что еще одним достоинством является тот факт, что данные в БД необходимо загружать только один раз. Впоследствии, имея несколько репозитариев в базе, довольно легко выполнять анализ кода с помощью одной команды.

3.7. QJ-Pro. Пакет QJ-Pro является продолжением работы компании QA-Systems, которая ведет разработку статических анализаторов с 1999 г. [13] (рис. 4).

Данный верификатор предназначен для проверки формата кода, соблюдения основных правил написания исходного текста. Пакет QJ-Pro имеет удобный и понятный графический интерфейс. В настройках проекта можно задать набор правил, тем самым упрощая анализ кода. Следует отметить, что пакет поставляется в собранном виде, где разработчики приготовили файлы для его запуска в средах Windows, Linux и Solaris, поэтому пользователь избавлен от необходимости работы в командной строке.

Анализ может быть выполнен для всего проекта (для отдельного пакета (package) в проекте либо для конкретного исходного java-файла). Результаты выводятся в виде таблицы с указанием номера строки и описанием замечания. При этом можно легко перейти к строке в исходном коде, к которой относится данное замечание. Интуитивно понятный интерфейс является одним из ключевых достоинств этого пакета. QJ-Pro идеально подойдет для начинающего разработчика, работа с ним не вызовет особых трудностей.

3.8. JDepend. Пакет предназначен для проверки зависимостей проекта и оценки оптимальности проектирования. Первоначально проект разрабатывался Робертом Мартином для анализа кода на языке C++, позже был переписан для анализа кода Java Майком Кларком [14]. Пакет можно запустить как в графическом, так и в консольном режиме (рис. 5).

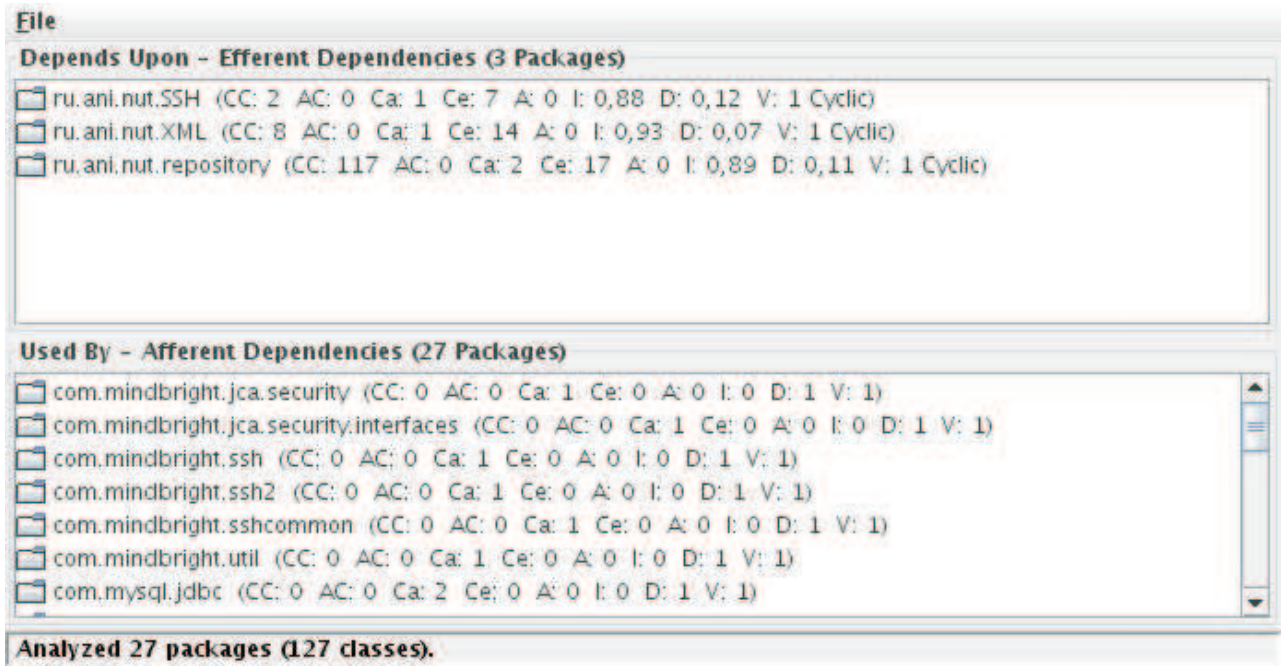

Рис. 5. Скриншот пакета JDepend

В интерфейсе отображаются все зависимости рассматриваемого проекта. В скобках указаны ключевые показатели, характеризующие гибкость, стабильность и расширяемость анализируемого кода.

3.9. Condenser. Пакет предназначен для поиска дублирующегося кода в программе [15]. С его помощью можно не только найти повторяющиеся фрагменты кода, но и автоматически удалить их. Программа использует технологию BCEL для работы с байт-кодом и jar-файлами. Отметим, что пользоваться автоматизированным удалением неиспользуемого кода нужно крайне осторожно. Например, при использовании Reflection (“отражение” — англ., динамическая загрузка, создание экземпляров классов, динамическое использование методов) некоторый код может явно не вызываться в программе, но в то же время его удаление приведет к ошибке.

3.10. Dependency Finder. Пакет Dependency Finder (рис. 6), предназначенный для анализа зависимостей разрабатываемого проекта, работает с байт-кодом [16]. Заметим, что можно воспользоваться как графическим интерфейсом пакета, так и запустить DependencyFinder в Web-режиме (для этого требуется загрузить war-файл (WAR — Web ARchive, архив веб-приложения) с сайта разработчика и запустить его на любом servlet-контейнере, например Tomcat или JBoss). Перед запуском пакета следует отредактировать файл web.xml в war-архиве, указав путь к папке, где расположены jar-файлы (либо .class-файлы)

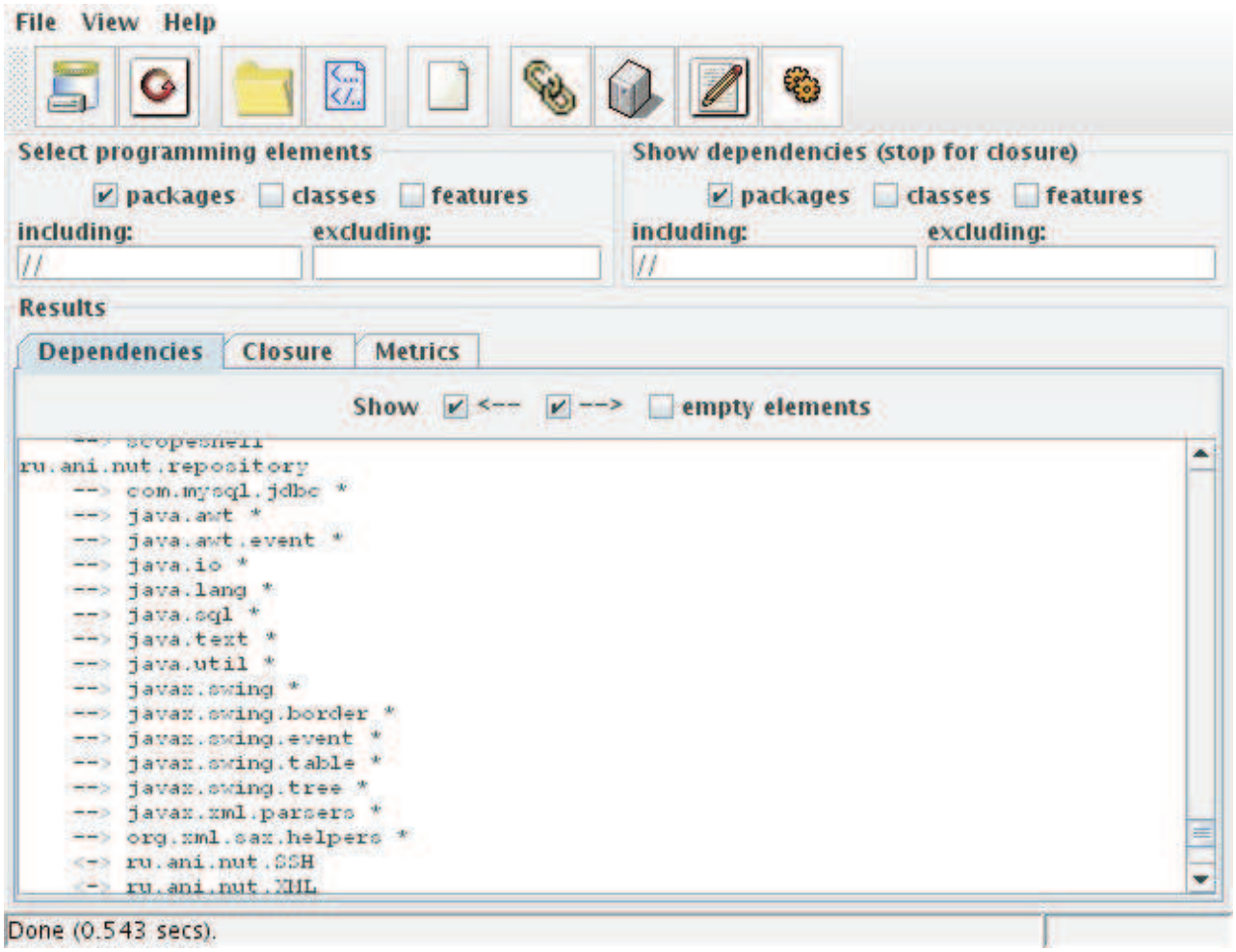

Рис. 6. Скриншот пакета Dependency Finder

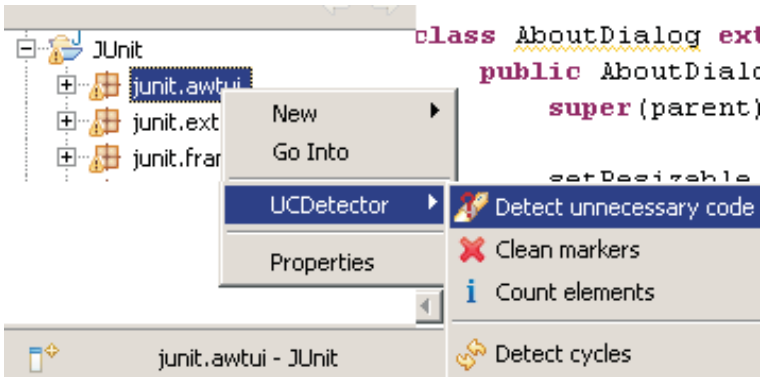

Рис. 7. Скриншот пакета UCDetector

для анализа. Для работы сначала нужно загрузить байт-код с помощью встроенной утилиты. После успешной загрузки можно запустить анализ зависимостей. Информация выводится в текстовом виде с указанием зависимостей для каждого пакета в анализируемом проекте. GUI-интерфейс проекта аналогичен web-интерфейсу. Для начала анализа нужно предварительно загрузить jar-файл программы (либо class-файлы), затем запустить анализ. В целом GUI-интерфейс является более простым и в то же время более подробным. Web-интерфейс может пригодиться при решении коллективом разработчиков более сложных задач.

3.11. UCDetector. Плагин для среды разработки Eclipse предназначен для поиска неиспользуемого в проекте кода [17]. Использование пакета не вызовет у пользователя проблем, UCDetector (рис. 7) интегрируется в среду Eclipse. Для поиска неиспользуемого кода достаточно щелчком правой кнопки мыши вызвать меню.

Заметим, что пакет следует использовать с осторожностью. Например, фреймворки Spring и Hibernate определяют использование классов в своих xml-конфигурационных файлах, но в коде этот же класс может явно не вызываться. Пользователь может отметить такие классы с помощью комментария “//NO_UCD”.

4. Сравнительный анализ. Нами был выполнен сравнительный анализ описанных выше средств

для статической верификации кода. Отметим, что пакеты довольно сильно отличаются между собой как по функциональности, так и по удобству использования.

В качестве объекта анализа использовалась программная система ScopeShell объемом более 35 тысяч строк, написанная на Java версии 1.4, поддерживающая работу с базой данных, многопоточную обработку данных, пользовательский интерфейс. Это позволяет более четко выявить возможности всех пакетов для статического анализа.

В табл. 1 представлены основные возможности различных статических верификаторов. Как следует из этой таблицы, все рассмотренные пакеты, за исключением FindBugs, обладают возможностью анализа исходного кода. Что касается исследования байт-кода, то только половина верификаторов могла его осуществлять. Большинство пакетов поддерживает все версии Java. В то же самое время ESC/Java работает с программами, написанными на Java версии 1.4 или более ранней (хотя у нас не возникало проблем при использовании Java 1.6). Несомненными достоинствами ряда пакетов является удобный графический интерфейс пользователя (FindBugs, QJ-Pro).

Таблица 1

Основные возможности пакетов

	GUI	Анализ байт-кода	Анализ исх. кода	Плагины для IDE	Экспорт результатов	Поддержка версий Java
FindBugs	+	+	—	+	+	Все
pmd	—	—	+	+	+	Все
ESC/Java	+	—	+	—	+	< 1.4
Lint4j	—	+	+	+	—	< 1.5
JLint	—	+	+	—	—	Все
Hammurapi	+	—	+	—	+	Все
QJ-Pro	+	—	+	—	+	Все

Результаты применения рассмотренных пакетов к программной системе ScopeShell приведены в табл. 2.

Таблица 2

Результаты применения верификаторов для анализа системы ScopeShell

	Общее кол-во ошибок	Несущественные ошибки	Потоки	Производительность	Серьезные ошибки	Безопасность
FindBugs	1449	256	103	304	374	27
Pmd	1949	1517	—	143	13	—
ESC/Java	1754 (734)	453 (211)	57 (46)	544 (281)	464 (191)	41 (41)
Lint4j	698	67	153	251	122	0
JLint	615	120	265	112	107	—
Hammurapi	1870	891	91	285	401	0
QJ-Pro	> 10000	> 10000	0	187	254	—

Рассмотрим расшифровку названий столбцов табл. 2.

1. “Общее количество ошибок” — количество ошибок и предупреждений, выданных анализатором в ходе проверки кода.

2. “Несущественные ошибки” — ошибки в написании исходного текста, отклонение от стандарта, применение операторов ==, != для String и др. Например, пакет FindBugs при анализе среды ScopeShell выявил такие ошибки, как использование метода System.exit(), предупреждая о том, что использование

этого метода может привести к ошибке в случае вызова другой программой, а также игнорирование возвращаемого некоторыми функциями результата, например при вызове delete() к объекту типа File.

3. “Потоки” — количество ошибок, связанных с синхронизацией потоков, возможными dead-lock’ами и др. В качестве примера можно привести использование метода Thread.start() в конструкторе некоторого класса, что может привести к проблемам при наследовании данного класса.

4. “Производительность” — ошибки, отрицательно влияющие на производительность (например, наличие неиспользуемых переменных, неэффективное использование конструкторов и др.). Анализаторы выявили в коде ScopeShell такие ошибки, как использование методов toString() у объектов типа String, конструкции if–else с одинаковым кодом в обеих ветвях, вызовы конструкторов new Integer(int) и new String(), что приводит к ненужному использованию памяти и снижению производительности программы.

5. “Серьезные ошибки” — такие ошибки, как повторное использование одних и тех же методов с одинаковыми параметрами, неверное преобразование типов, ненужные проверки, неверная обработка Exception, ошибки в switch() и др. Примером может стать отсутствие строки break в условиях switch(), что зачастую приводит к ошибке. Пакеты pmd, findBugs, ESC/Java выявили довольно много потенциальных ошибок в коде ScopeShell, связанных с отсутствием проверок на null, часть из них действительно приводила к нарушению функционирования программы при определенных условиях. В качестве примера можно привести ошибку в операторе switch(), которая приводит к нарушению логики программы:

```
int result = tableManager.validateRow(currentRow);
switch(result) {
      case -1 : JOptionPane.showMessageDialog(this, "Ошибка, в поле введен недопустимый
      символ");
      case 0 : tableManager.saveRow(currentRow);
      break;
      case 1: JOptionPane.showMessageDialog(this, "Требуемые поля не заполнены");
}
```

По идее, при возвращаемом коде, равном -1, программа должна выдать предупреждение, но из-за отсутствия break в первом условии, программа выдаст предупреждение, но все же сохранит неверные данные.

6. “Безопасность” — ошибки, которые могут привести к появлению уязвимостей в программе. В коде ScopeShell были найдены ошибки при формировании SQL запроса конкатенацией строк без каких-либо проверок, что является серьезной уязвимостью. Из-за конкатенации строки запроса появляется возможность применить так называемую SQL-инъекцию для несанкционированного доступа к данным в БД. В качестве примера можно рассматривать следующий код:

```
Connection conn = DriverManager.getConnection(url, Properties.username, Properties.password);
identRow flRow = rowManager.getRow(rowId);
String sql= "INSERT into files (files_id, repo_id, file_name)
VALUES ("+ flRow.flId +",’"+    flRow.flReposId + "’,’" +
flRow.flName"’)";
PreparedStatement cstmt = conn.prepareStatement(sql);
boolean result = cstmt.execute(sql);
```

В этом фрагменте кода используется конкатенация строки запроса к БД. Данный код не вызывает ошибок при работе программы, но представляет серьезную угрозу для безопасности БД.

Для пакета pmd приведены результаты при использовании следующих правил анализа: basic, braces, naming и unusedcode. Результаты могут значительно измениться при использовании других правил либо создании собственных. Для пакета ESC/Java результаты даны без использования JML. После добавления комментариев JML в исходный код количество ошибок значительно сокращается. В скобках приведены результаты анализа исходного кода с использованием JML.

С помощью пакета QJ-Pro проводился анализ с полным набором проверок, поэтому общее количество предупреждений более 10 000. Бóльшая часть предупреждений — рекомендации по оформлению кода. Тем не менее, пакет смог обнаружить серьезные ошибки в коде. Для упрощения анализа полученной информации следует ограничить набор проверок при анализе.

Отдельного упоминания требует пакет ESC/Java. Для начала пакет был проанализирован на исходном коде без добавления каких-либо комментариев на языке JML. Затем были внесены необходимые прагмы JML и выполнен повторный запуск пакета ESC/Java. Отметим, что с помощью применения JML удалось добиться значительного упрощения и повышения эффективности анализа кода, а именно уменьшения в два раза общего количества предупреждений. Однако добавление комментариев на языке JML

для большого проекта потребует значительных усилий, а также подробного изучения документации по использованию различных прагм. Отметим также, что нам не удалось добиться стабильной работы графического интерфейса, поэтому анализ проводился в консольном режиме, что довольно сильно затрудняло работу.

Отдельно были рассмотрены пакеты для построения и анализа зависимостей. Здесь следует отметить пакет JDepend. Для анализа JDepend предоставляет набор показателей, приведенных в табл. 3.

Таблица 3

Основные показатели пакета JDepend

Код	Описание	Значение
CC — Concrete class count	Общее количество классов в пакете	310.0
AC — Abstract class	Количество abstract-классов и интерфейсов	0.0
Ca — Afferent Couplings	Количество пакетов, зависящих от классов в анализируемом пакете	27.0
Ce — Efferent Couplings	Количество пакетов, от которых зависит анализируемый пакет	3.0
A — Abstractness	Отношение количества abstract-классов и интерфейсов к сумме общего количества классов и интерфейсов (AC/(CC+AC))	0.0
I — Instability	Ce/(Ce+Ca)	0.1
Cyclic	1, если в пакете существует циклическая зависимость, 0 в противном случае	1.0

Заметим, что значения приведены для основного пакета среды ScopeShell, отвечающего за создание интерфейса и основные функции.

Так, показатель A (Abstractness) для графической среды ScopeShell составил 0.0, что является показателем затрудненной расширяемости пакета. Показатели Ca и Ce составили 27.0 и 3.0 соответственно, что говорит о том, что в целом код стабилен относительно изменений в других пакетах, но в то же время изменения в исходном коде среды ScopeShell могут привести к ошибкам в других пакетах. Соответственно, показатель I (Instability) равен 0.1, что подтверждает очень высокую стабильность пакета.

Пакет Dependency Finder удобен для более простого анализа зависимостей. С его помощью не удалось получить таких важных оценок, как в JDepend, однако Dependency Finder имеет простой и информативный интерфейс и оказался более удобным при поверхностном анализе.

Пакеты Condenser и UCDetector смогли выявить в коде ScopeShell несколько неиспользуемых методов. Еще раз отметим, что этими пакетами нужно пользоваться с особой осторожностью.

5. Заключение. В статье рассмотрены свободно распространяемые программные средства для статической верификации кода, написанного на языке Java. Наибольшей проблемой при их использовании является большое количество ложных предупреждений. В то же самое время с их помощью мы смогли выявить более двухсот ошибок. С определенной уверенностью можно сказать, что подобные пакеты следует использовать при разработке даже небольших пакетов. Сложно выделить какой-то один пакет вследствие их значительных различий в работе и предназначении. Однако, по нашему мнению, для начинающего разработчика лучшим вариантом окажется использование FindBugs и QJ-Pro как наиболее простых и в то же время достаточно развитых пакетов для статической верификации. Для опытного пользователя более интересным будет применение pmd и Hammurapi, являющихся более гибкими пакетами с возможностью создания собственных правил анализа, а также ESC/Java с поддержкой JML языка. В дальнейшем для верификации ScopeShell мы планируем воспользоваться так называемым микромодельным подходом, представляемым в рамках проекта Alloy [18, 19]. Модели Alloy — это сравнительно небольшие модели, отражающие наиболее важные аспекты архитектуры программной системы. Они могут быть автоматически проанализированы с помощью Alloy Analyzer [20].

СПИСОК ЛИТЕРАТУРЫ

1. Gnuplot (http://www.gnuplot.info/).
2. *Hovemeyer D., Pugh W.* Finding bugs is easy // ACM Sigplan Notices. 2004. **39**. 92–106.
3. FindBugs (http://findbugs.sourceforge.net).
4. BCEL (http://jakarta.apache.org/bcel).
5. PMD/Java (http://pmd.sourceforge.net).
6. Lint4j (http://www.jutils.com).
7. JLint (http://artho.com/jlint).
8. *Artho C.* Finding faults in multi-threaded programs. Master's Th. Institute of Computer Systems, Federal Institute of Technology. Zürich/Austin, 2001.
9. *Flanagan C., Leino K.R.M., Lillibridge N.M.G., Saxe J.B., Stata R.* Extended static checking for Java // Proc. of the 2002 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Berlin, 2002. 234–245.
10. *Burdy L., Cheon Y., Cok D., Ernst M., Kiniry J., Leavens G.T., Leino K.R.M., Poll E.* An overview of JML tools and applications // Int. J. on Software Tools for Technology Transfer. 2005. **7**, N 3. 212–232.
11. *Leavens G.T., Leino K.R.M., Poll E., Ruby C., Jacobs B.* JML: Notations and tools supporting detailed design in Java // OOPSLA'00 Companion. Minneapolis, 2000. 105–106.
12. Hammurapi (http://www.hammurapi.biz).
13. QJ-Pro (http://qjpro.sourceforge.net).
14. Clarkware Consulting (http://www.clarkware.com).
15. Condenser (http://condenser.sourceforge.net).
16. Dependency Finder (http://depfind.sourceforge.net).
17. UCDetector (http://www.ucdetector.org).
18. *Jackson D.* Micromodels of software: modelling and analysis with Alloy (http://sdg.lcs.mit.edu/alloy/book.pdf).
19. Alloy (http://alloy.mit.edu).
20. *Jackson D., Schechter I., Shlyakhter I.* ALCOA: The Alloy constraint analyzer // Proc. of the 22nd Int. Conf. on Software Engineering. Limerick, 2000. 730–733.

Поступила в редакцию
27.11.2008