

Анализ покрытия как метод улучшения качества кода

Ольга Бегунова (InterComponentWare AG), Алексей Валиков
(Forschungszentrum Informatik)

1. Введение

В целях оптимизации процесса разработки и повышения качества кода необходимо иметь инструмент, позволяющий осуществлять автоматический контроль качества. Одним из подходов к решению проблемы оценки качества кода является использование принципов архитектуры, управляемой тестами (test-driven architecture, TDA).

[Юнит-тесты](#) позволяют проверить, действительно ли отдельная часть кода выполняет именно те действия, которых ожидает разработчик, и не делает ничего другого, для чего тестируются очень маленькие, изолированные части кода. При этом остается открытым вопрос, отвечает ли результат выполнения кода ожиданиям клиентов или конечных пользователей. На этапе написания юнит-тестов не проводится валидация, верификация и проверка правильности конечного результата.

Кроме того, юнит-тесты помогают проверить, как код обрабатывает при различных входных данных. Таким образом, юнит-тесты играют роль исполняемой документации. Такого рода исполняемая документация всегда актуальна, потому что юнит-тесты должны обновляться с изменением основного кода. Кроме того, юнит-тесты позволяют остальным членам команды разработчиков просмотреть примеры использования кода.

Некоторые программисты до сих пор считают, что они не должны писать юнит-тесты, поэтому все еще необходима мотивация разработчиков к написанию тестов для их кода.

Мотивация может стимулироваться политическими методами, такими как, например, введение написания юнит-тестов как обязательного этапа разработки. Однако такое решение не будет самым удачным, поскольку качество обязательных к написанию тестов будет далеким от совершенства. Другим, более мягким методом мотивации является индивидуальная оценка качества кода, выполненная для каждого отдельного разработчика. Одним из способов оценки качества является анализ покрытия кода тестами при помощи специальных инструментов. Помимо того, что инструменты анализа покрытия кода дают достаточно точные показатели качества проводимого тестирования, результаты анализа могут быть представлены в виде индивидуальных отчетов о качестве покрытия кода для каждого разработчика. В этом случае разработчика не заставляют писать юнит тесты, однако сравнение индивидуальных показателей служит стимулом к написанию более подробных тестов, что в свою очередь приводит к улучшению качества кода.

2. Создание индивидуальных отчетов качества кода

Результатом работы большинства существующих инструментов анализа покрытия кода является общий отчет, показывающий, какие части программы были выполнены в процессе прогона тестов, а какие — нет. Чтобы вычислить индивидуальные показатели для каждого разработчика, требуется перейти к индивидуальным отчетам. Для этого необходимо выяснить, кто из разработчиков отвечает за определенный компонент программы. Таким образом, задача получения отчета о покрытии кода тестами для каждого разработчика при ближайшем рассмотрении разбивается на три этапа.

На первом этапе посредством существующих инструментов анализа создается отчет о покрытии тестами всего проекта.

На втором этапе составляется актуальный список распределения областей ответственности разработчиков (list of responsibilities).

И, наконец, отчет о покрытии обрабатывается на основании описания областей ответственности.

По результатам проведенного анализа строится таблица, в которой приводятся индивидуальные показатели для каждого разработчика.

Анализ покрытия как метод улучшения качества кода

Анализ покрытия кода позволяет обнаружить секции кода, которые не были в достаточной мере протестированы юнит-тестами. Если юнит-тесты улучшают качество кода, то инструменты анализа покрытия кода в свою очередь улучшают качество юнит-тестов. Отчеты о покрытии кода могут быть предоставлены в различных форматах, что значительно облегчает последующий анализ данных. Анализируя полученные таким образом отчеты, разработчик составляет подробную картину того, что его тесты на самом деле тестируют, а какие части кода остаются необработанными. Такая обратная связь способствует увеличению мотивации разработчиков к написанию тестов.

При разработке приложений большого и среднего масштабов сложно отслеживать список распределения ответственности разработчиков. В лучшем случае существует список пакетов приложения с указанием отвечающего за пакет разработчика. При этом отдельные модули пакета могут создаваться другими программистами. Кроме того, в процессе разработки ответственный разработчик может меняться, что не всегда вовремя отражается в документации.

Для составления индивидуальных отчетов необходим более подробный и точный список распределения ответственности разработчиков. Необходимы точные данные о том, какой разработчик отвечает за каждый отдельный класс проекта в текущий момент времени. Получить такой список для проектов большого и среднего масштаба достаточно сложно. Часто даже ответственный за пакет разработчик не может быстро предоставить полный список файлов входящих в пакет, а обработка и упорядочивание этих данных требует неоправданных временных затрат.

Дополнительной сложностью является то, что список областей ответственности должен быть оформлен в формате, допускающем дальнейшую обработку (например, в XML).

На третьем этапе на основе полученного отчета о покрытии кода тестами и детального описания областей ответственности создаются индивидуальные отчеты для каждого разработчика.

Для этого необходимо произвести сортировку отчета о покрытии кода по авторам и найти средний коэффициент покрытия кода для каждого разработчика.

Важным требованием ко всем трем этапам является возможность автоматического

выполнения всего процесса.

3. Реализация

В данном разделе приводится практическое решение задачи автоматического создания индивидуальных отчетов о покрытии кода тестами с применением инструмента анализа покрытия кода Clover и инструмента построения проекта [Apache Ant](#).

[Clover](#) — один из наиболее популярных инструментов анализа покрытия кода для Java. Он предоставляет возможность измерять покрытие кода тестами поэлементно, а не построчно, как большинство других предлагаемых сегодня инструментов анализа. Clover позволяет проводить анализ покрытия кода на уровне методов, операторов и выражений для всего проекта или его отдельных пакетов, файлов или классов.

Перед компиляцией Clover специальным образом помечает проверяемые элементы и создает обработанную копию java-кода. Созданная копия исходного кода компилируется обычным способом. В процессе компиляции информация обо всех артефактах в исходном коде сохраняется в базу данных Clover. Таким образом, при компиляции кода с Clover создается специальная база данных, содержащая метки на все проверяемые Clover элементы кода.

При выполнении юнит-тестов Clover калибрует созданную базу данных и проверяет, сколько раз был выполнен каждый отдельный элемент кода при прогоне тестов. Полученные данные Clover записывает в обновленную базу данных, которая обрабатывается в дальнейшем при составлении отчетов.

Clover легко интегрируется в билд Apache Ant включением clover.jar в библиотеки Ant и определением задачи clovertasks в файле build.xml:

```
<taskdef resource="clovertasks" classpathref="ant.classpath"/>
```

Для создания базы данных покрытия используется задача <clover-setup>.

Для включения Clover в процесс компиляции в файл build.xml необходимо добавить целевую задачу (target) Ant 'with.clover':

```
<target name="with.clover">
  <mkdir dir="clover-db"/>
  <clover-setup initstring="clover-db/ant_coverage.db">
```

Анализ покрытия как метод улучшения качества кода

```
</clover-setup>  
</target>
```

Для создания базы данных покрытия необходимо откомпилировать билд с Clover при помощи команды:

```
ant with.clover default-build.
```

(default-build — целевая задача, при помощи которой мы компилируем обычный билд для Ant.)

Для получения базы данных покрытия кода тестами после компиляции билда при помощи Clover необходимо запустить целевую задачу Ant для выполнения всех тестов:

```
ant all_tests.
```

В результате прогона тестов данные о покрытии кода будут записаны в базу данных покрытия clover-db\ant_coverage.db.

Задача Ant для создания отчета Clover <clover-report> также описывается в файле build.xml.

Для создания отчета о покрытии в формате XML выполняется целевая задача 'clover_xml':

```
<target name="clover_xml" depends="with.clover">  
  <clover-report>  
    <current outfile="coverage.xml">  
      <format type="xml"/>  
    </current>  
  </clover-report>  
</target>
```

При выполнении этой задачи Ant полученный отчет о покрытии кода сохраняется в файле coverage.xml.

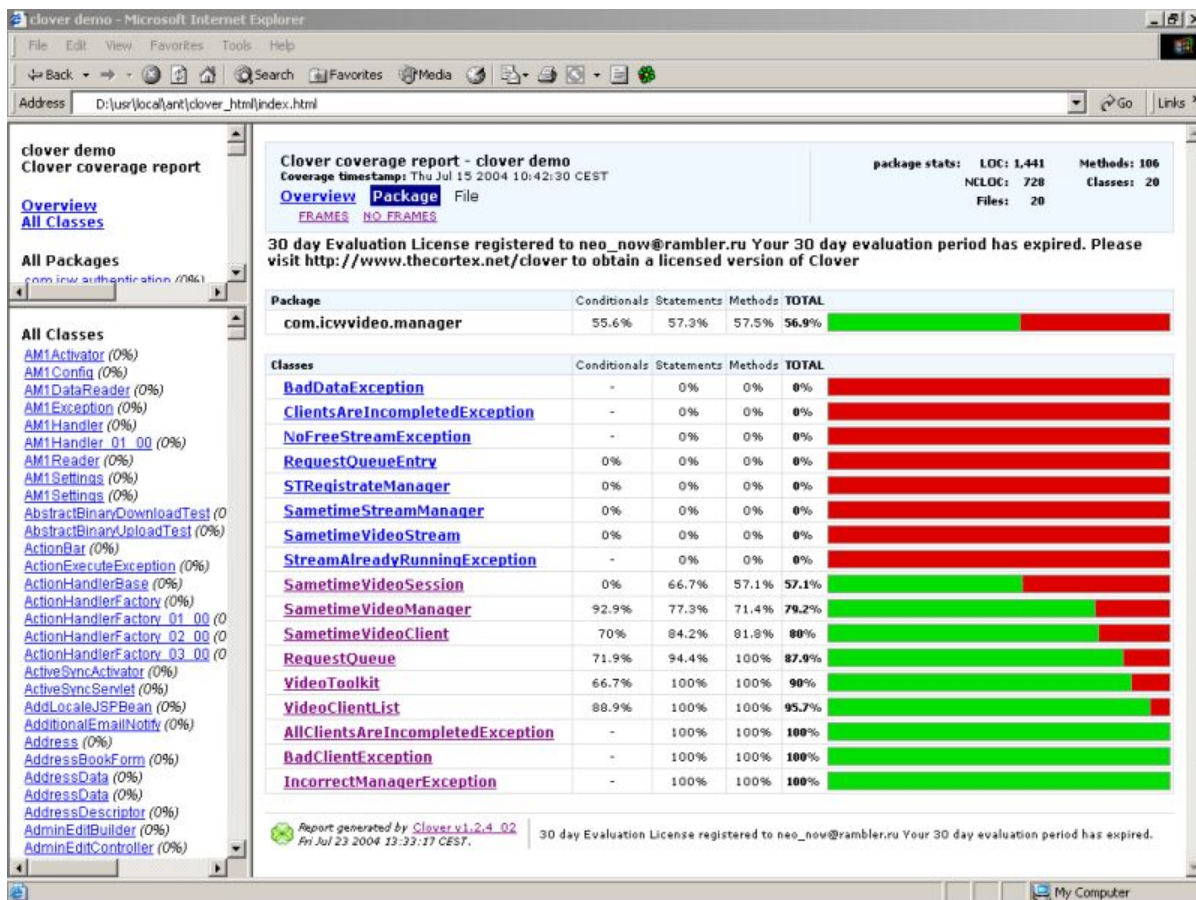
Для получения отчета в формате HTML используется целевая задача 'clover_html':

```
<target name="clover_html" depends="with.clover">  
  <clover-report>
```

Анализ покрытия как метод улучшения качества кода

```
<current outfile="clover_html" title="clover demo">
  <format type="html"/>
</current>
</clover-report>
</target>
```

Полученный отчет в формате HTML можно просмотреть при помощи web-браузера.



Для получения актуального списка ответственности разработчиков используются данные [CVS](#) (Concurrent Versions System tool). Ответственным за файл системы принято считать разработчика, который добавил в CVS последнюю версию файла. Таким образом, для получения списка ответственности необходимо получить данные о последнем изменении файлов проекта. Для этого используется целевая задача Ant [<CvsChangeLog>](#).

Анализ покрытия как метод улучшения качества кода

При помощи `<CvsChangeLog>` Ant генерирует отчет об изменениях в репозитории CVS и выводит его в XML формате:

```
<target name="cvschangelog">
  <cvschangelog dir="../../Source "
               destfile="log/cvslog.xml">
  </cvschangelog>
</target>
```

Для создания индивидуальных отчетов полученные XML-файлы (отчет по покрытию и список изменений в CVS) обрабатываются при помощи [XSLT-преобразований](#) (eXtensible Stylesheet Language for Transformation).

Для автоматического выполнения XSLT-преобразований из Ant в последних версиях Java (1.4.x и старше) можно применять задачу [mtxslt](#). Эта задача поддерживает множество XSLT-процессоров, включая Saxon 7, реализующий XSLT 2.0. В более ранних версиях Java можно остановить свой выбор на процессоре Xalan-Java, вызывая его как обычную Java-программу.

Для получения индивидуальных отчетов о покрытии последовательно применяются несколько XSLT-преобразований.

Первое XSLT-преобразование анализирует CVS log-файл и выбирает для каждого файла последнюю ревизию с наиболее поздней датой:

```
<xsl:template match="files">
  <responsibilities>
    <xsl:apply-templates
      select="file[generate-id(.)=generate-id(key('file',@name))]" />
  </responsibilities>
</xsl:template>
<xsl:template match="file">
  <xsl:variable name="files" select="key('file', @name)" />
  <xsl:variable name="file" select="$files[@timestamp=dyn:max($files,@timestamp)]" />
  <responsibility file="{ $file/@name}" author="{ $file/@author}" />
</xsl:template>
```

В результате преобразования мы получаем неотсортированный список распределения областей ответственности разработчиков по всем файлам в следующем формате:

Анализ покрытия как метод улучшения качества кода

```
<responsibility author="Ivanov" file="Source/com/action/ActionResponse.java"/>
<responsibility author="Petrov" file="Source/com/action/Handler.java"/>
```

Для вызова XSLT-преобразования из Ant необходимо добавить в файл Ant classpath путь к библиотеке xalan.jar.

В файл build.xml добавляется целевая задача 'xslt'

```
<target name="xslt ">
  <java classname="org.apache.xalan.xslt.Process" fork="true" dir="${basedir}">
    <arg value="-in" />
    <arg value="file-revisions.xml" />
    <arg value="-out" />
    <arg value="responsibilitylist.xml" />
    <arg value="-xsl" />
    <arg value="responsibilities.xsl"/>
  </java>
</target>
```

Следующее преобразование обрабатывает отчет Clover, представленный в формате XML в файле coverage.xml, и файл responsibilitylist.xml со списком распределения областей ответственности разработчиков, и создает неотсортированный список покрытия java-файлов с указанием ответственного разработчика:

```
<xsl:template match="coverage">
  <files>
    <xsl:apply-templates select="project/package/file"/>
  </files>
</xsl:template>
<xsl:template match="file">
  <xsl:variable name="name" select="translate(substring-after(@name, $dir), '\', '/')">
  <xsl:if test="metrics/@elements > 0">
    <file name="{ $name}" coverage="{metrics/@coveredelements div metrics/@elements}"
      coveredelements="{metrics/@coveredelements}" elements="{metrics/@elements}"
      author="{ $responsibilities[@file=$name]/@author}"/>
  </xsl:if>
</xsl:template>
```

В результате преобразования создается отчет о покрытии java-кода с указанием имени

Анализ покрытия как метод улучшения качества кода

ответственного разработчика следующего вида:

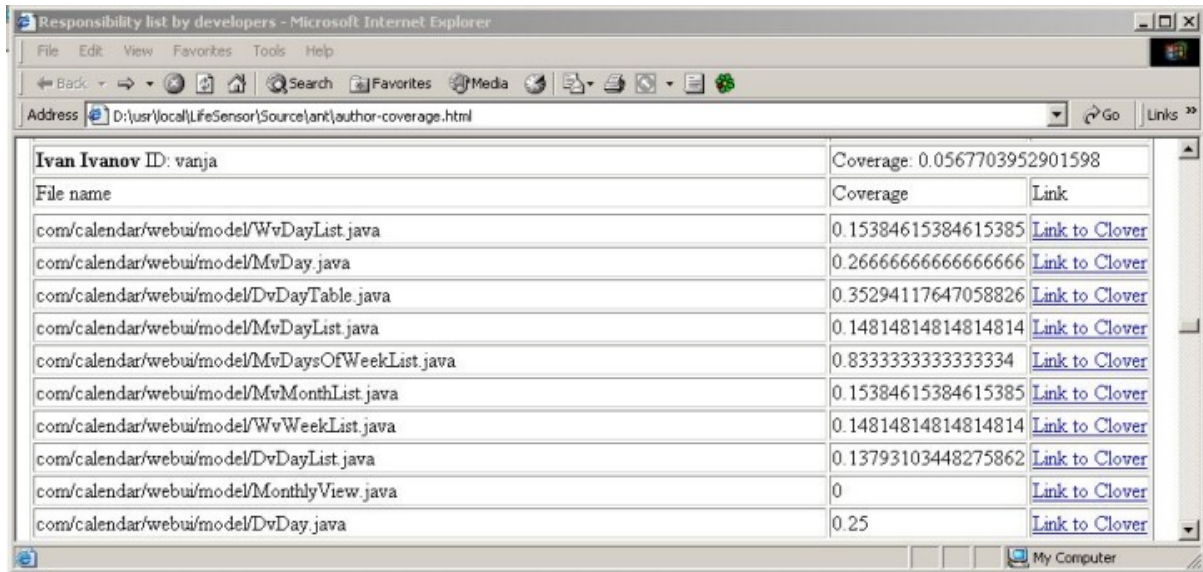
```
<files>
...
<file author="Ivanov" coverage="0.7916666666666666"
  name="Source/com/ Sample.java"/ elements="72" coveredelements="57">
<file author="Petrov" coverage="0.8787878787878788"
  name="Source/com/RequestQueue.java"/ elements="99" coveredelements="87">
<file author="Smirnov" coverage="0.5714285714285714"
  name="Source/com/SampleSession.java"/ elements="42" coveredelements="24">
...
</files>
```

Следующее XSLT-преобразование сортирует отчет о покрытии по разработчикам и находит среднее значение покрытия кода для каждого разработчика:

```
<xsl:template match="files">
  <authors>
    <xsl:apply-templates
      select="file[generate-id(.)=generate-id(key('file',@author))]" />
  </authors>
</xsl:template>
<xsl:template match="file">
  <xsl:variable name="list" select="key('file', @author)" />
  <author id="{@author}" name="{ $users[@id=current()/@author]/@name}"
    averagecoverage="{sum($list/@coveredelements) div sum($list/@elements)}">
    <xsl:copy-of select="$list" />
  </author>
</xsl:template>
```

Полученный XML файл легко преобразуется в HTML формат.

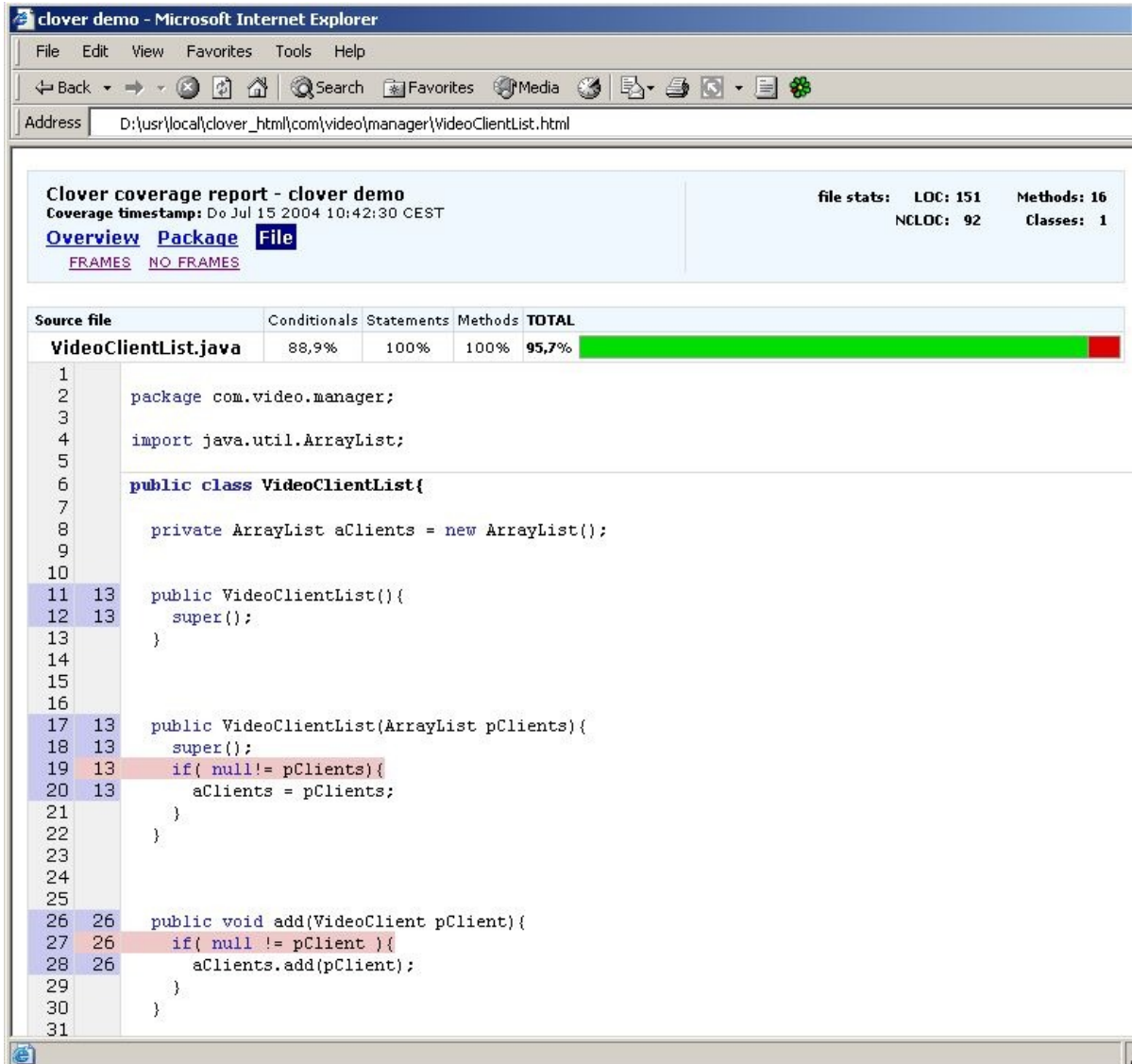
Анализ покрытия как метод улучшения качества кода



File name	Coverage	Link
com/calendar/webui/model/WvDayList.java	0.15384615384615385	Link to Clover
com/calendar/webui/model/MvDay.java	0.26666666666666666	Link to Clover
com/calendar/webui/model/DvDayTable.java	0.35294117647058826	Link to Clover
com/calendar/webui/model/MvDayList.java	0.14814814814814814	Link to Clover
com/calendar/webui/model/MvDaysOfWeekList.java	0.8333333333333333	Link to Clover
com/calendar/webui/model/MvMonthList.java	0.15384615384615385	Link to Clover
com/calendar/webui/model/WvWeekList.java	0.14814814814814814	Link to Clover
com/calendar/webui/model/DvDayList.java	0.13793103448275862	Link to Clover
com/calendar/webui/model/MonthlyView.java	0	Link to Clover
com/calendar/webui/model/DvDay.java	0.25	Link to Clover

Добавление ссылок на отчеты Clover, представленные в формате HTML, позволяет переходить к детальной информации по покрытию того или иного класса.

Анализ покрытия как метод улучшения качества кода



Clover coverage report - clover demo
Coverage timestamp: Do Jul 15 2004 10:42:30 CEST

file stats: LOC: 151 Methods: 16
NCLOC: 92 Classes: 1

[Overview](#) [Package](#) [File](#)
[FRAMES](#) [NO FRAMES](#)

Source file	Conditionals	Statements	Methods	TOTAL
VideoClientList.java	88,9%	100%	100%	95,7%

```
1
2 package com.video.manager;
3
4 import java.util.ArrayList;
5
6 public class VideoClientList{
7
8     private ArrayList aClients = new ArrayList();
9
10
11 13 public VideoClientList(){
12 13     super();
13
14 }
15
16
17 13 public VideoClientList(ArrayList pClients){
18 13     super();
19 13     if( null!= pClients){
20 13         aClients = pClients;
21
22     }
23
24 }
25
26 26 public void add(VideoClient pClient){
27 26     if( null != pClient ){
28 26         aClients.add(pClient);
29
30     }
31 }
```

4. Заключение

В данной статье приводится способ решения задачи повышения мотивации разработчиков к тестированию кода посредством получения индивидуальных оценок качества проводимых тестов. Для этого при помощи специализированных инструментов выполняется общий анализ покрытия кода, результаты которого

представляются в виде индивидуальных отчетов для каждого разработчика.

Решение задачи увеличения мотивации рассматривается на примере реализации автоматического процесса генерации индивидуальных отчетов о качестве тестирования с применением инструмента анализа покрытия Clover. Для получения текущего списка распределения областей ответственности разработчиков используется журнал изменений системы контроля версий CVS. Обработка отчетов выполняется при помощи XSLT-преобразований.

Предлагаемый подход позволяет повысить мотивацию программистов к написанию тестов, что положительным образом сказывается на качестве программного обеспечения. Такой метод мотивации не является политическим или административным, следовательно, будет с большей готовностью принят разработчиками.

Описанный метод также удобно использовать для перехода к методологии разработки, управляемой тестами (англ test-driven development, [TDD](#)).

Кроме того, индивидуальные оценки являются важными показателями для руководителей групп и менеджеров проектов, позволяющими составить достаточно точные характеристики персонала.

5. Ресурсы

- [JUnit](#) - JUnit, Testing Resources for Extreme Programming;
- [Clover](#) - Cenqua Clover Code Coverage for Java;
- [Apache Ant](#) - Java-based build tool;
- [XSLT](#) - eXtensible Stylesheet Language for Transformation (см. также [спецификацию](#));
- [mtxslt](#) - Multi-XSLT Ant Task;
- [CVS](#) - Concurrent Versions System;
- [CvsChangeLog](#) - целевая задача Ant <CvsChangeLog>;
- [TDD](#) - Test-driven development.