|  |
| --- |
| CodePro AnalytiX – анализ и разбиране на големи софтуерни продукти на Java |
|
| *Проект по „Софтуерни технологии“, 2015 г.* |
| Факултет математика и информатика, Софийски университет |

|  |
| --- |
| Изготвили: |
| *80866*, *Стефан Василев Василев*, *thesopa@abv.bg*  *Компютърни науки, четвърти курс, втори поток, група 8* |
| *80855*, *Драгомир Светославов Тунчев*, *staticbg@gmail.com*  *Компютърни науки, четвърти курс, втори поток, група 8* |
| *80867*, *Антонио Найденов Николов*, *antonionikolov93@gmail.com*  *Компютърни науки, четвърти курс, първи поток, група 4* |
| *80851*, *Росен Тодоров Рачев*, *rosen\_ra4ev@abv.bg*  *Компютърни науки, четвърти курс, втори поток, група 8* |

Ръководител: доц. Димитър Биров

2015 г.

Съдържание

[1 Увод 3](#_Toc437253207)

[1.1 Тема на проекта 3](#_Toc437253208)

[1.2 Цели на проекта 3](#_Toc437253209)

[1.3 Резултати 4](#_Toc437253210)

[2 Описание 4](#_Toc437253211)

[2.1 Изисквания към софтуера и инсталиране 5](#_Toc437253212)

[2.1.1 Инсталация 5](#_Toc437253213)

[2.2 Структура на инструмента, основни компоненти, подкомпоненти 7](#_Toc437253214)

[2.2.1 Code analysis 7](#_Toc437253215)

[2.2.2 JUnit test generation 11](#_Toc437253216)

[2.2.3 junit test editor 13](#_Toc437253217)

[2.2.4 audit rule categories 15](#_Toc437253218)

[2.2.5 metrics 18](#_Toc437253219)

[2.2.6 metric categories 19](#_Toc437253220)

[2.2.7 dependency analysis 20](#_Toc437253221)

[2.2.8 javadoc maintenance 23](#_Toc437253222)

[2.2.9 code coverage 24](#_Toc437253223)

[3 Използване на CodePro Analytix 27](#_Toc437253224)

[3.1 Auditing 27](#_Toc437253225)

[3.2 Прилагане на Audit върху трите тестови проекта и разглеждане на резултата 29](#_Toc437253226)

[3.2.1 Retrofit 29](#_Toc437253227)

[3.2.2 Rxjava 29](#_Toc437253228)

[3.2.3 Elasticsearch 31](#_Toc437253229)

[4 Мнение на авторите 34](#_Toc437253230)

[Библиография 34](#_Toc437253231)

# Увод

С напредъка на софтуерните технологии се появяват все повече и повече големи продукти. Те сами по себе си съдържат изключително много файлове, пълни с програмен код и зависимости, които е трудно да бъдат съобразени от обикновения програмист. Ето защо на помощ идват инструменти за автоматизирана проверка на коректността на продукта.

## Тема на проекта

В този проект ще разгледаме конкретния инструмент CodePro AnalytiX. С помощта на CodePro AnalytiX можем лесно да управляваме софтуерните си продукти, тъй като той предлага набор от средства за верификация на програмния ни код, както и коректността на зависимостите между файловете ни. По този начин се спестява изключително много време и усилия на софтуерния разработчик. Ето защо е удачно инструмента да се използва за особено големи проекти, в които има ясно установени срокове за изпълнение на конкретни задачи. Освен това автоматизацията на процеса на проверка на програмния код предотвратява възможността за допускане на тривиални грешки, като объркани имена на файлове, пакети и други, които често човек може да обърка при невнимание.

## Цели на проекта

На първо място целта е да се научим да работим в екип, всеки от нас да поема отговорност за собствената си задача, но да не забравя да се консултира за това и с останалите членове на екипа. Освен това повечето от нас са стажанти в момента и всеки ден се сблъскваме с големи проекти и голямо количество програмен код. Ето защо никога не е излишно да се запознаем с още един добър инструмент за анализ на код и управление на големи проекти. Сложните алгоритми, които стоят зад един инструмент за анализиране на код също представляват голям интерес и могат да ни бъдат полезни в бъдещи проекти. И на последно място в крайна сметка е добре да обогатяваме общата си култура в сферата на готовите софтуерни продукти и плъгини, които могат да ни бъдат в ползва когато сами правим своите проекти. С усилията, които положихме да се запознаем с CodePro AnalytiX научихме и доста странични и полезни неща.

## Резултати

С присъствие на презентацията на този проект, зрителят може да:

* Се запознае с плюсовете, които ни дава възможността да анализираме кода си автоматично.
* Се убеди, че инструментите за анализ на код са изключително полезни за големи софтуерни продукти.
* Се научи да инсталира инструменти за анализ на код, както и други различни плъгини в редактора Eclipse.
* Се запознае по-подробно с инструмента CodePro AnalytiX. С неговата функционалност, възможности, кога и как е удачно да бъде използван от софтуерния разработчик.
* Види конкретно приложение на CodePro AnalytiX.

Главната методология за съставяне на проекта беше проучване в интернет за материали, видео, указания за работа с продукта. Също така четене, тестване и разбиране на документацията на CodePro AnalytiX.

# Описание

Като всеки софтуерен продукт и CodePro AnalytiX се нуждае от определена конфигурация на компютъра, както и от редица други зависимости и особености, които трябва да изпълнява нашата машина, за да може да бъде инсталиран.

## Изисквания към софтуера и инсталиране

На първо място компютърът ни трябва да има инсталирана операционна система. Тя може да бъде Windows или Linux.

### Инсталация

#### Стъпка 1 – Сваляне на редактора Eclipse

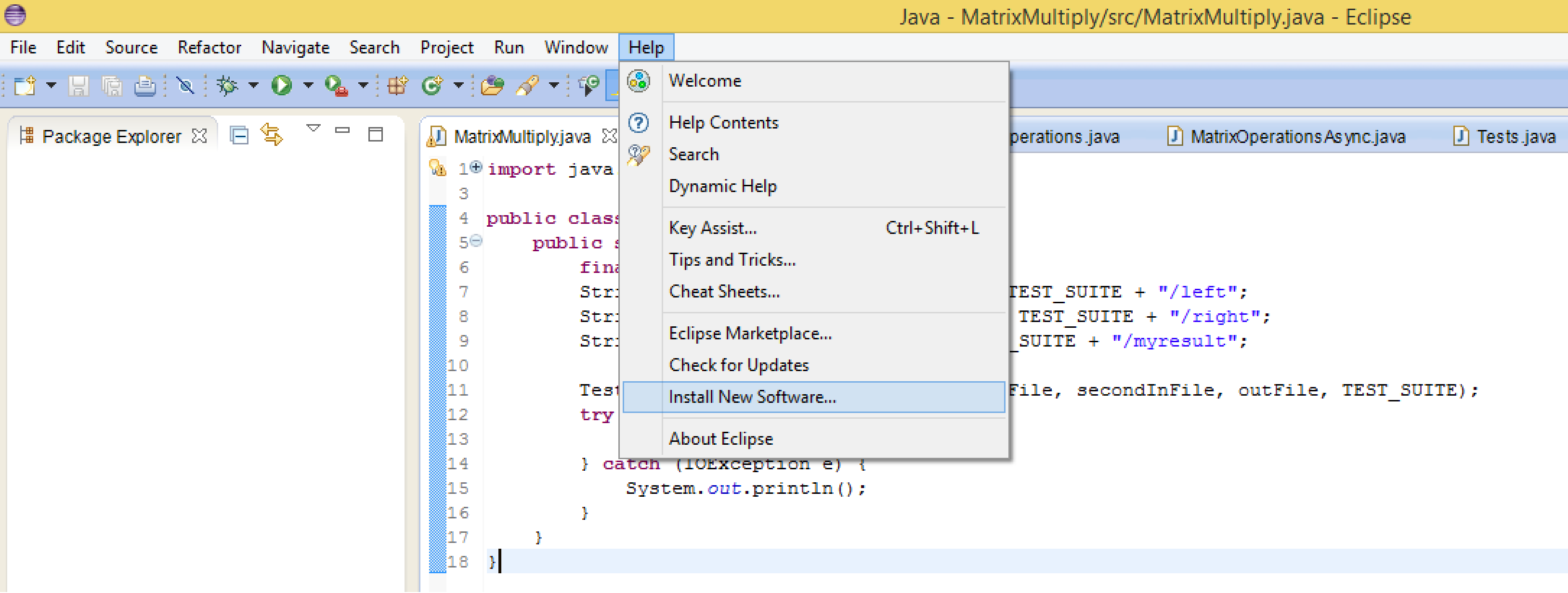
След като има операционна система потребителят трябва да изтегли версия на Eclipse за Java разработчици (препоръчително най-новата) от официалния сайт:

http://www.eclipse.org/downloads/

#### Стъпка 2 – Инсталиране на CodePro AnalytiX

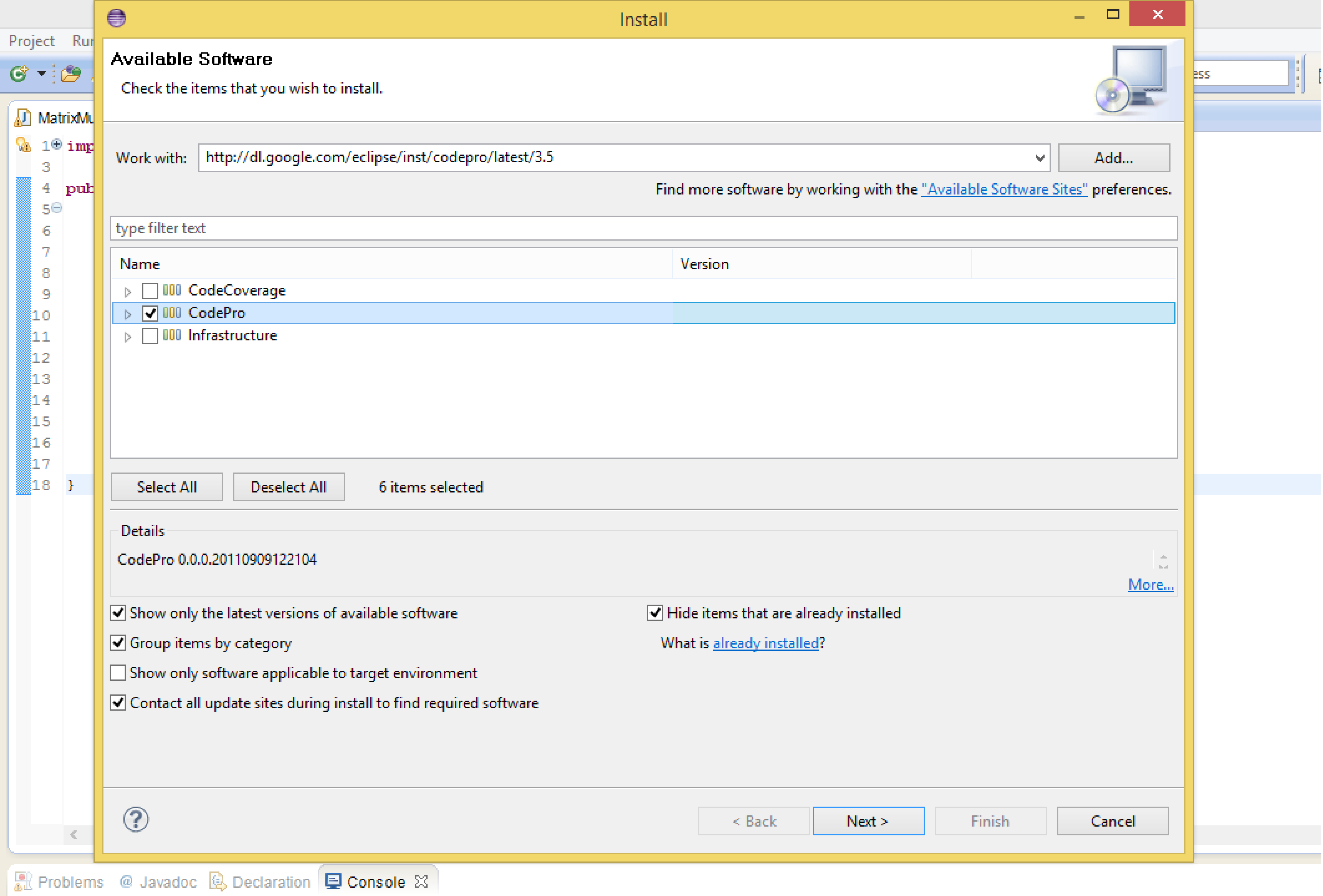
След инсталация на редактора Eclipse вече сме готови да инсталираме CodePro AnalytiX. Тъй като платформата на Eclipse работи главно с плъгини, самия редактор предоставя лесен начин за инсталиране на нови плъгини.

След като стартираме Eclipse, можем лесно да се навигираме до падащото меню Help -> Install New Software…



**Фиг.1.** *Инсталиране, стъпка 1*

В резултат на това се отваря прозорец, в който трябва да въведем линк, от който Eclipse ще може да свали CodePro AnalytiX. Тъй като има значение коя версия на редактора сме изтеглили, трябва да отидем на официалния сайт на CodePro и да вземем линк за сваляне на плъгина, съответстващ на нашата версия на Eclipse. След като въведем адреса за сваляне в прозореца имаме няколко опции. Чекваме опцията CodePro:



**Фиг.2.** *Инсталиране, стъпка 2*

След това кликваме два пъти Next, прочитаме и се съгласяваме с условията за използване на плъгина и избираме Finish. Изчакваме известно време, след което рестартираме Eclipse (инсталацията сама ще ни помоли да го направим), за да могат промените да влязат в действие.

## Структура на инструмента, основни компоненти, подкомпоненти

CodePro Analytix съдържа в себе си множество инструменти за различен тип анализ на програмен код. Целта е да се подобри качеството му. Това се постига, като се анализира кода и инструментът открива ненужни усложнения, грешки и др.

Инструментите за анализ на кода могат да бъдат разглеждани като компоненти на CodePro Analytix.

### Code analysis

#### Code Audit

Чрез Code Audit се прави проверка на програмния код дали отговаря на дадени правила. Инструмента съдържа много вградени правила, но могат да бъдат добавяни и нови такива.

#### Audit Explorer

Audit Explorer-ът показва ресурсите, които са създадени от и свързани с Code Audit функционалността. Два вида ресурси се показват: Audit Result Set – показват се нарушенията, установени при проверката; Audit Series – показва се отчет, базиран на серия от Audit Result Set-ове.

#### Audit Rule Sets

Тук се показва избраният набор от правила за проверка, както и всички достъпни правила за проверка.

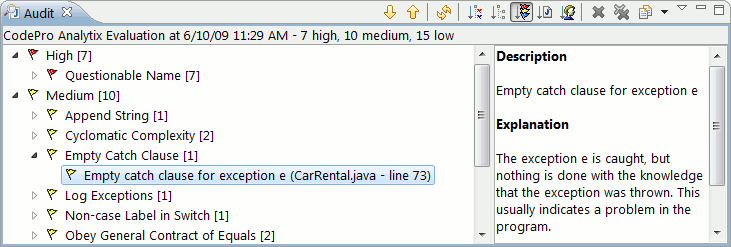
#### Dynamic Code Auditing

Възможно е да се изпълни динамичен одит на някои пакети (специфицирани от включващи и изключващи пакети). Когато е включен динамичният одит за даден пакет, отварянето на клас в този пакет резултира в автоматично сканиране за одит нарушения в този клас и резултатите се добавят в Audit изгледа във Violations in Open Files множеството от одит нарушения.

Динамичният одит на код е изключително полезен за откриване на проблеми докато програмистът пише код, вместо след като кодът вече е написан. Едно нарушение в именуването, например, може да бъде пренесено в много файлове преди вечерен build, където одитът би открил нарушение. Динамичният одит отбелязва проблема в най-близкият възможен момент от момента получаване на проблема. Цената за оправяне на проблеми в писането на код често е право пропорционална а времето между създаване на проблема и момента на неговото откриване. Динамичният код одит е еквивалентен на това да имате личен код reviewer, който стои до Вас през цялото време и Ви съобщава за проблеми при нужда.

#### Audit View

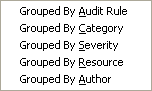
Audit изгледът съдържа списък от нарушения на правилата за одит, които са се получили от пускането на тези одит правила върху целевите елементи. Името на всяко одит правило с нарушение е показано в корена на дървото. Разтварянето на кое да е одит правило показва списък от индивидуални нарушения с техните локации. Избрани нарушения могат да бъдат премахнати от множежоството и да се копират в clipboard-а.



**Фиг.3.** *Audit View*

#### Generating Reports

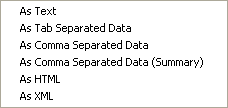
Generate Report опцията позволява да се запазят резултатите от одит във HTML report. Резултатите могат да бъдат групирани по одит правило, категория на одит правило, тежест, ресурс или автор.



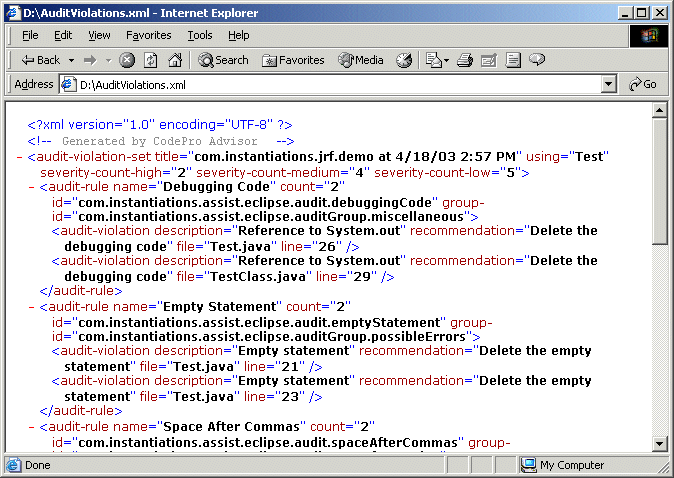
**Фиг.4.** *Audit Menu Generate Report*

#### Exporting Result Data

Export Violations опцията позволява да се запазят резултатите от одит директно в текстов файл в текст, разделен с табове, разделен със запетаи (пълен репорт или обобщение), HTML или XML формат. Чрез задържане на Ctrl на клавиатурата, могат да се запазят само избрани нарушения от всички.



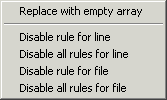
**Фиг.5.** *Export Report Menu*



**Фиг.6.** *Audit Report XML*

#### Fixing Violations

Fix Violation командата показва списък от решения на нарушенията, които са предложени от audit facility. Избиране на решение прилага решението върху избрани нарушения. В някои случаи, игнорирането на нарушения е правилният избор на действие. Затова са предоставени и няколко опции за локално изключване на правилата. Някои решения може да се приложат едновременно на няколко нарушения, докато други могат да се прилагат само на едно нарушение.



**Фиг.7.** *Fix Violation Menu*

### JUnit test generation

#### Test Case Generation

CodePro JUnit Test Case Generation позволява автоматизирано създаване на обширни JUnit тестове за регресия. По даден входен клас, инструментът създава съответстващ тест клас, пълен с множество тест методи за всеки метод от входния клас. Инструментът анализира всеки метод и входните аргументи с цел да генерира тестове, които обхващат всеки ред от програмния код.

#### Design by Contract поддръжка

Един начин да се промени Java кода с цел подобряване на качеството на генерирания код е да се включи design by contract информация. Design by contract (дизайн по договор) е подход за разработка на софтуер, който се формализира в езика Eiffel. Идеята е да се направи експлицитен договор между клиентския код (кодът, който ще извиква някакво API) и кодът, който се извиква. По-конкретно, искаме да уточним: какво трябва да бъде вярно за една инстанция на клас (инварианта), за да бъде в консистентно състояние; какво трябва да бъде вярно преди даден метод да може да бъде извикан (предпоставка), което включва както състоянието на обекта, така и ограниченията над стойностите на аргументите; какво трябва да бъде вярно след като методът е приключил, което включва както състоянието на обекта, така и ограниченията над върнатата стойност.

В Java, поддръжката на design by contract има две форми. Първата е чрез използването на допълнителни Javadoc тагове. Този подход не е стандартизиран, но има няколко конвенции, които са по-популярни от останалите и CodePro AnalytiX поддържа по-известните форми. Втората е чрез използване на assert изявление. Тя е достъпна в JDK 3.0 и нататък. Освен това, assert изявленията могат да бъдат използвани и по други причини, освен да изразяват договори и интерпретацията на assert изявление не е винаги очевидна. Приемаме, че assert изявления, които се намират в началото на метод, представляват предпоставки, а assert изявления, които се намират в края на метод, представляват ограничения над връщаната стойност на метода, въпреки че те могат да представляват и инварианти, но CodePro AnalytiX не може да ги различи.

Чрез настройките, които са достъпни през CodePro -> JUnit -> Design by Contract потребителят може да контролира как се борави с design by contract информацията.

Има три основни вида тагове: инварианти, предпоставки и постусловия. При всеки от тези видове, тагът се следва от текст, който трябва да бъде в един от следните формати:

      <booleanExpression>

      ( <booleanExpression> , <messageExpression> )

Инвариантите се появяват вътре в Javadoc за клас. Има две обичайни форми за инвариантните тагове - @invariant и @inv, като и двете се разпознават. Кодът в инвариантен таг може да се отнася към кои да е видими типове и към кои да е полета или методи, дефинирани от конкретния клас. Потребителят може да специфицира дали инвариантите трябва да бъдат проверявани във всеки тестов метод, но това не е задължително.

Например, ако имаме клас, който представлява стек от цели числа, в който стойностите в стека са запаметени в масив с име elements с integer поле с име index, чиято стойност е индекса на слота след главата на стека, може да напишем инварианта по следния начин:

      @invariant (index >= 0) && (index <= elements.length)

Както предпоставките, така и постусловията се появяват вътре в Javadoc за методи и конструктори. Предпоставките са представени чрез @pre тага; постусловията са представени чрез @post тага. Кодът в предпоставките може да реферира към всичко, към което може инварианта да реферира, както и към параметрите за метода или конструктора, с изключение, че предпоставките за конструктори не могат да реферират полета или методи на инстанци – само статични елементи.

Например, ако пишем клас, който представя работник и искаме да осигурим, че името на работника е винаги непразен символен низ, може да добавим предпоставка към setName метода, чийто параметър е символен низ с име newName, по следния начин:

      @pre newName != null && newName.length() > 0

В допълнение, кодът за постусловия мжое да съдържа два специални текста. Текстът $result може да бъде включен към постусловие за всеки метод с тип на връщаната стойност, различен от void, за да реферира към резултатът от извикване на метода. Текст във формата

      $pre ( <type> , <expression> )

също може да бъде добавен. Този текст се интерпретира като референция към стойността на израз, който трябва да бъде от даден тип, преди извикването на метода. Като пример, нека имаме метод, който увеличава стойността на поле с име counter с 1. Този метод може да има постусловие във формата:

      @post counter == $pre(int, counter) + 1

Потребителят може да специфицира дали постусловията трябва да бъдат проверявани във всеки тестов метод за дадения целеви метод, но това не е задължително.

#### Test Suite Generation

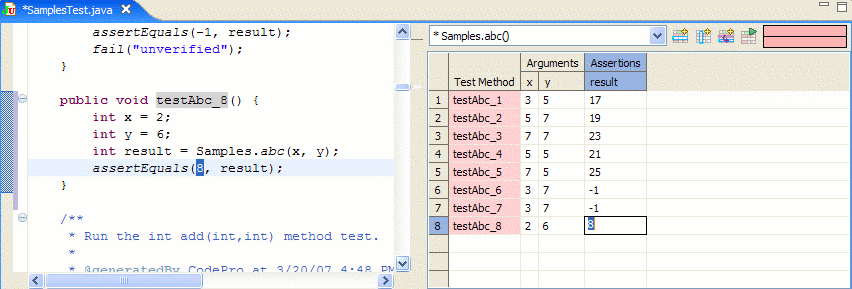
Потребителят може да избере дали тест суити да бъдат генерирани, и ако да, дали да автоматично да включват в себе си тестовете от тест суитите дефинирани в подпакети. Потребителят може също да избере името на тест суитите.

### junit test editor

Чрез едитора могат да се добавят тестове и assertion-и.

#### Adding Tests

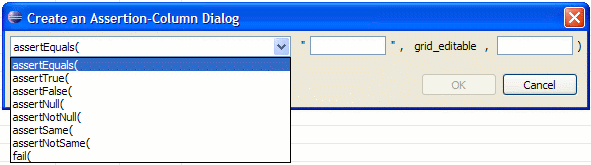
Тест методи могат да бъдат добавяни чрез Тест редактора или чрез директно писане в Java редактора: За да добавим тест метод, използвайки Тест редактора, избираме Add Test Method бутона или натискаме Ctrl + Shift + N. Нов тестови метод се добавя в списъка. Въвеждаме нужните аргументи и assert изявления и запазваме работата си. Новият тестови метод вече трябва да се вижда и в Java редактора. Може да се добавят нови тестови методи директно в Java редактора и те автоматично се появяват в Тест редактора след запазване.



**Фиг.8.** *Добавяне на тест*

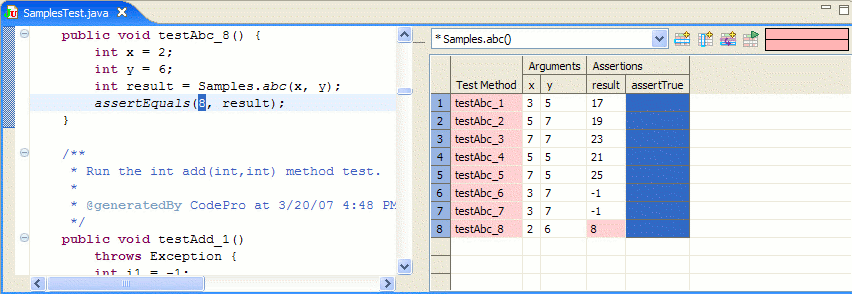
#### Adding Assertions

Допълнителни assert изявления могат да бъдат добавени или като директно се пише в Java редактора или като се използва Тест редактора: За да добавим assert изявление, използвайки Тест редактора, избираме Add Assertion бутона или натискаме Ctrl + Shift + A. Появява се диалог, който приема входни данни както те излизат в кода. Въвеждаме нужните входни данни и натискаме OK.



**Фиг.9.** *Create Assertion Column Dialog*

Празна колона е добавена към мрежата, която се държи като всички останали колони.



**Фиг.10.** *Assert Columns*

### audit rule categories

#### Coding Style

Група от правила, които търсят програмен код, който не отговаря на зададен стил на писане на код.

#### Comments

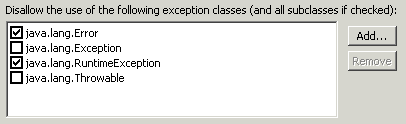
Група от правила, които проверяват правилното използване на коментари, които не са от тип Javadoc.

#### Dead Code

Група от правила, които търсят код, който не се използва или не може да бъде достигнат (мъртъв код).

#### Exceptions

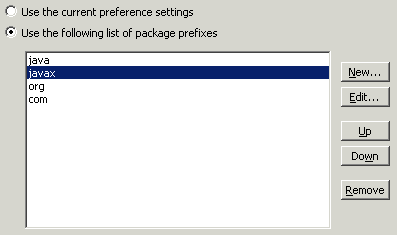
Група от правила, които проверяват за проблеми, свързани с употребата на изключения в програмния код.



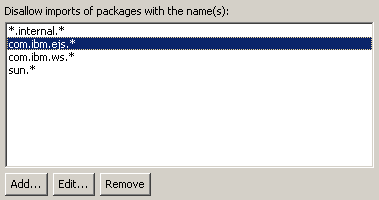
**Фиг. 11**. *Забраняване на класове изключения*

#### Import Usage

Група от правила, които проверяват дали импортите отговарят на общоприетите практики за писане на код.



**Фиг. 12**. *Избиране на префиксите, с които могат да започват импортите*



**Фиг. 13**. *Забраняване на импорти на класове с въведените в списъка имена*

#### Inheritance

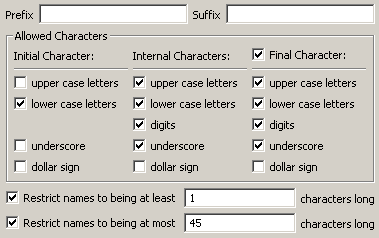
Група от правила, които проверяват за проблеми, които са свързани с класовата йерархия или могат да бъдат видяни само при преглеждане на суперкласовете на клас.

#### Performance

Група от правила, които откриват използването на практики за писане на код, които могат да доведат до проблеми с производителността. Тези правила не откриват всеки източник на проблеми, нито всичко открито от тях е реален проблем за производителността.

#### Naming Conventions

Група от правила, които проверяват имената на различни програмни елементи дали отговарят на въведените стандарти и конвенции.



**Фиг. 14.** *Настройки за стандартите на именуване на елементи*

#### Security

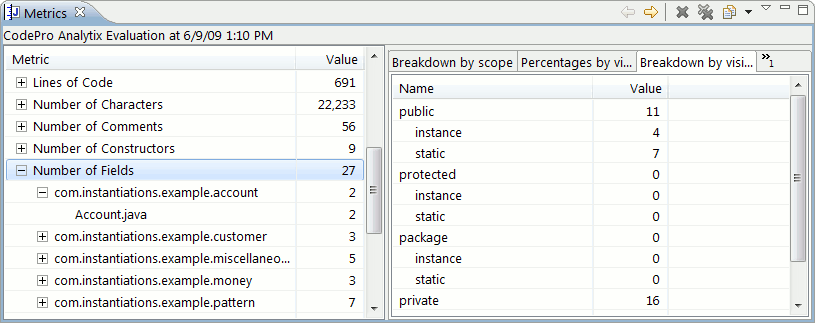
Група от правила, които откриват използването на практики за писане на код, които водят до по-несигурен код.

### metrics

При избиране на "Compute Metrics" от менюто, включените измервания в множеството измервания по подразбиране (default metric set) биват пуснати. Това създава множество от резултати от измервания (metric results set), което се показва в Metrics view екрана. Други множества от измервания могат да бъдат пускани чрез командата "Compute Metrics Using...".

Осигурени са опции в предпочитанията, които пускат или спират различни измервания, както и такива, които слагат различни прагове на допустими нарушения.

Таблицата на измерванията съдържа списък от измерванията, които са били пуснати и резултатите от тяхното пускане върху дадени елементи. Името на измерването се показва в първата колона, а стойността – във втората колона.



**Фиг. 15**. *Таблица на измерванията*

### metric categories

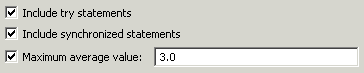
Измерванията са разделени и групирани по различни категории.

#### Basics

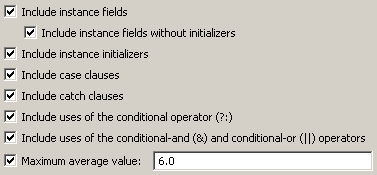
Група от измервания, които изчисляват основна информация за програмния код – Среден брой редове код на метод, Среден брой конструктори на клас, Среден брой полета на клас, Среден брой методи на клас, Среден брой параметри, Редове код, Брой символи, Брой коментари, Брой конструктори, Брой полета, Брой редове, Брой методи, Брой пакети, Брой точки и запетаи, Брой класове.

#### Complexity

Група от измервания, които оценяват сложността на програмния код.



**Фиг. 16**. *Настройки за измерванията на statement-и*



**Фиг. 17**. *Настройки за измерванията на програмни елементи*

#### Dependency

Група от измервания, които измерват отговорността, самостоятелността и стабилността на тяло програмен код.

#### Inheritance

Група от измервания, които са базирани на наследствената структура на програмния код.

#### Ratio

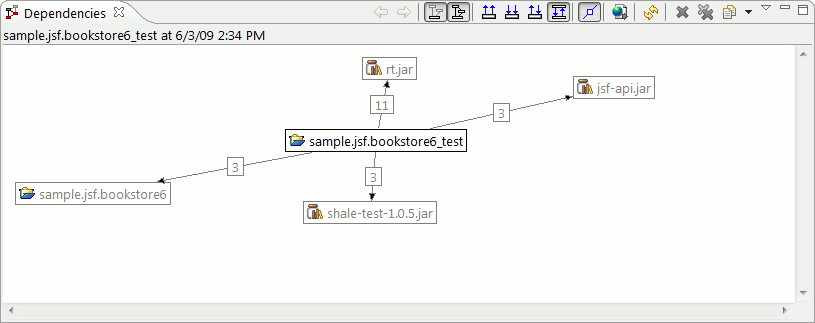
Група от измервания, които са базирани на отношението на един вид нещо към друго.

### dependency analysis

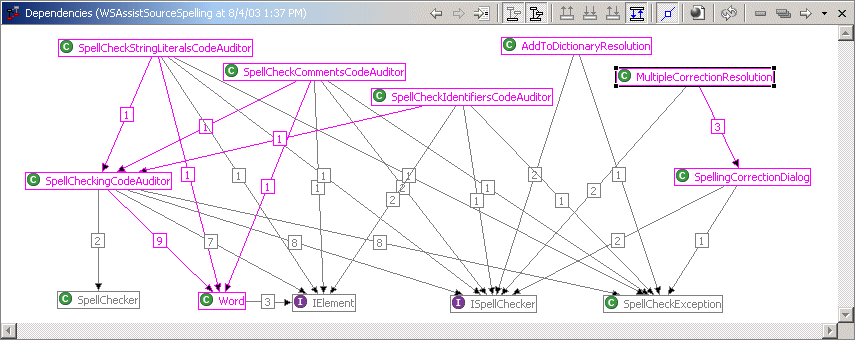
#### Dependencies

Екранът на зависимостите показва резултатите от анализ на зависимостите и може да бъде използван да се разграничат два проекта или пакета. Анализ на зависимостите може да се направи като се изберат един или повече проекти, корени на пакетни фрагменти или пакетни фрагменти в Package Explorer и после се избира Analyze Dependencies от подменюто CodePro Tools в контекстното меню. Системата поддържа списък от достъпни анализи на зависимостите.

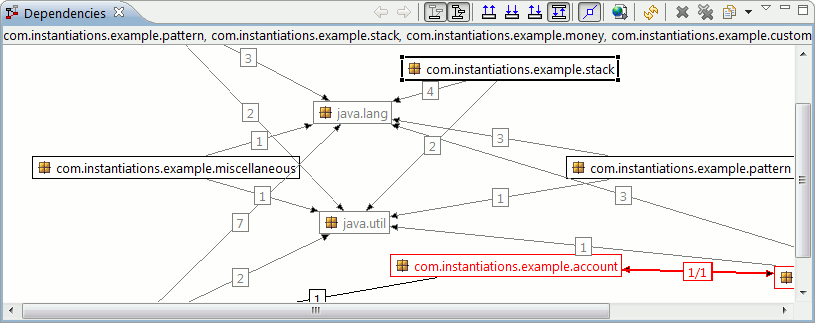
Зависимостите, които са открити могат да бъдат погледнати на три различни нива на детайлност – зависимости между проекти, пакети или класове. Ако анализът е изпълнен върху един или повече проекти, то всички нива на детайлност са достъпни. Ако анализът е изпълнен върху един или повече корени на пакетни фрагменти или пакетни фрагменти, то само нивата на зависимости между пакети и класове са достъпни. Когато се избере анализ се показва най-високото възможно ниво на детайлност.



**Фиг. 18**. *Зависимости с ниво на детайлност зависимости между проекти*



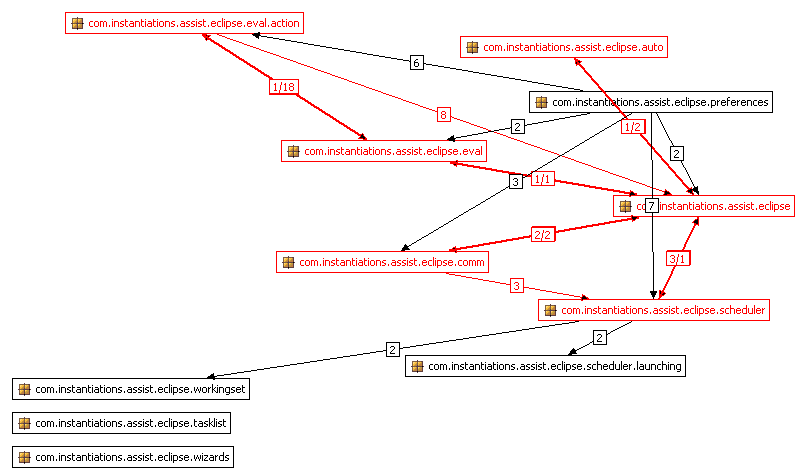
**Фиг. 19**. *Зависимости с ниво на детайлност зависимости между класове*



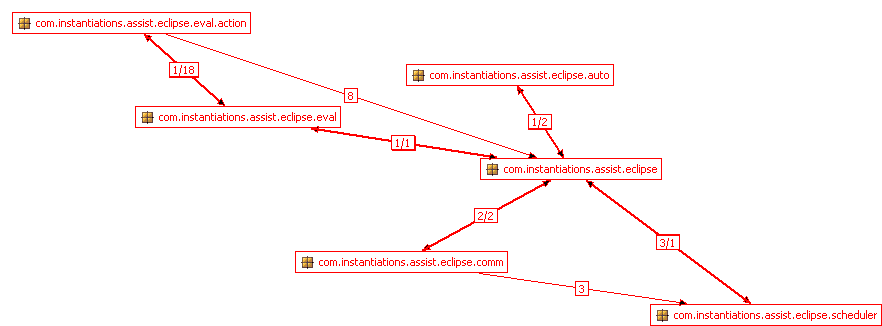
**Фиг. 20**. *Зависимости с ниво на детайлност зависимости между пакети*

#### HTML Report

Създава отчет за анализа на зависимостите в HTML формат.



**Фиг. 21**. *Граф на зависимостите между пакети*



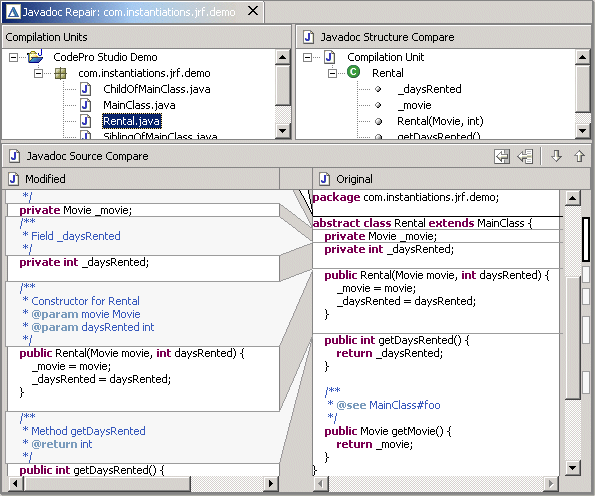
**Фиг. 22**. *Цикличност на пакетите*

### javadoc maintenance

#### Javadoc Repair

При натискане избиране на опцията "Repair Javadoc" от менюто се включва javadoc repair engine-ът.

Той открива липсващи Javadoc коментари, както и възможни корекции и подобрения на вече съществуващи такива. Резултатите се показват в Javadoc редактор за сравнение. В редактора, предложените промени могат да бъдат редактирани, приложени или отказани.



**Фиг. 23.** *Дървото на компилационните единици показва компилационните единици в избрания обхват, за които има предложени Javadoc промени*

### code coverage

#### Code Coverage

Code Coverage инструментите измерват каква част от програмния код се изпълнява. Най-често се използват, за да оценят ефективността на тестовете в преминаването през всички възможни пътища в кода.

Осигурени са два различни механизма. Изборът кой от двата е правилен за дадени нужди зависи основно от сложността на кода, който се пуска. Първият е направен за използване от самостоятелни (stand-alone) Java програми, които вървят върху единствена виртуална машина и не използват специализирани клас loaders.

Вторият механизъм е направен за какви да е програми, но може да изисква малко повече работа, за да се настрои първоначално, което отново зависи от колко точно сложен е кода. И двата механизма са разгледани в по-големи детайли по-долу.

#### Simple Code Coverage

Можем да измерим покритието на кода за дадена част от кода като изберем клас, който дефинира main метод и после изберем "Run Code Coverage" от "CodePro Tools" подменюто. Избраният клас ще се пусне и резултатите ще бъдат показани в Coverage View. От самото Coverage View, резултатите могат също да бъдат изкарани в различни формати, включително HTML, XML и текст.

Информацията за това как класът трябва да се пусне и какъв тип информация за покритието трябва да се събере е вкарана в специален вид launch конфигурация. Може launch конфигурацията да се създаде ръчно, но "Run Code Coverage" от менюто прави този процес по-лесен.

След като launch конфигурация е създадена, може да се измери покритието на кода или чрез "Run Code Coverage" опцията от менюто или като се пусне директно launch конфигурацията от "Run" менюто.

#### Full Featured Code Coverage

Code coverage работи като оркестрира байт кодовете, които се зареждат във виртуалната машина, за да събере данни относно кои байт кодове са се изпълнили. Пускането на кода чрез използване на code coverage launch конфигурация е оркестрирано динамично, тъй като се зарежда чрез специализиран клас loader. Този клас loader става клас loader по подразбиране за виртуалната машина, който се изпълнява първоначално. Ако допълнителни клас loaders са създадени от програмата или ако допълнителни виртуални машини са пуснати, те няма да изпълнят динамичното оркестриране, което е нужно за да се събере информацията за покритието на кода.

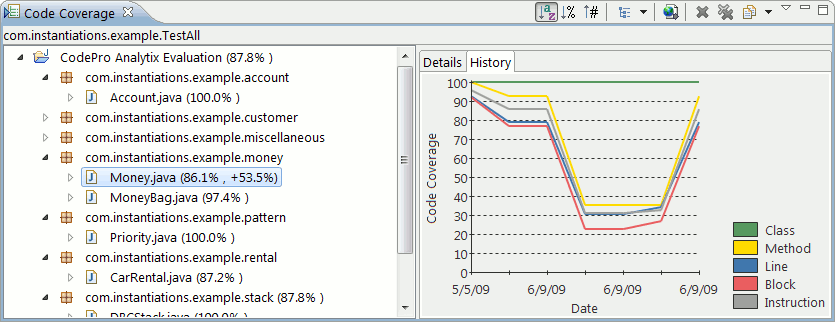
Алтернативата е да се оркестрира кода статично преди изпълнение на програмата. CodePro доставя поддръжка за оркестриране на кода на база проект-за-проект. Всеки проект има страница за Code Coverage свойства. Тази страница позволява да се контролира дали класовите файлове да бъдат оркестрирани. Също така, позволява да се контролира кои класови файлове са оркестрирани. Като удобство, "Instrument Code" и "Uninstrument Code" опциите в "CodePro Tools" подменюто позволяват включване и изключване на оркестрацията, като не променят останалите настройки свързани с проекта.

След като проектите са оркестрирани, може да се събира информация за покритието на кода по всяко време, когато е пуснат кода, като се използва коя да е launch конфигурация, която сме създали. Няма нужда и не трябва да се използва Code Coverage launch конфигурацията, за да се пуска преоркестриран код.

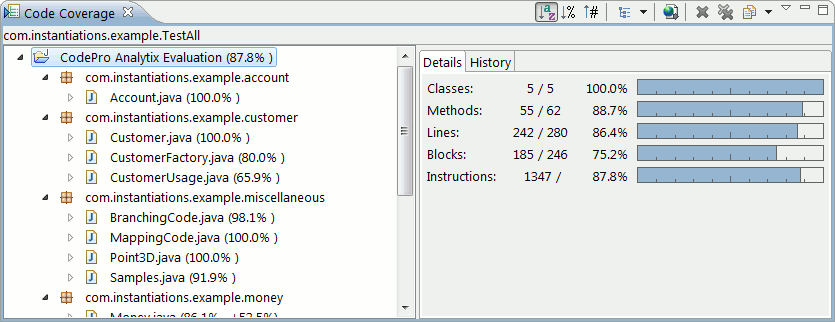
Има две други усложнения от използването на този механизъм, на които трябва да се обърне внимание. Първото е, че ако искаме да пуснем кода без да събираме данни за покритието му, трябва да премахнем оркестрирането на всички проекти, които съдържат кода, който искаме да пуснем. Второто усложнение е, че ако искаме да дистрибутираме клас файловете от локалната машина, трябва внимателно да премахнем оркестрирането на кода преди да го пакетираме за дистрибуция. В противен случай данните за покритието на кода ще бъдат събирани при всяко пускане на кода.

#### Coverage View

Coverage View екранът е разделен на две секции – съдържание и детайли. Съдържанието показва Java елементите, за които е записана информация за обхват на кода. Тези елементи могат да бъдат показани в йерархия или като списък от елементи. При избиране на елемент, в детайлите се показва информация за този елемент. Детайлите съдържат две страници, под формата на табове – детайли и история.



**Фиг. 24**. *Таб история*



**Фиг. 25**. *Таб детайли*

Страницата с детайлите показва информация за обхвата на класове, методи, редове код, блокове код и инструкции. Информацията се показва както като дроб – обхванати програмни единици разделени на общия брой програмни единици, така и като проценти. Процентите се показват и графично.

#### HTML Report

Създава отчет за обхвата на програмния код в HTML формат.

Code Coverage Summary 92,3% coverage

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Line Coverage** | 92.3% | | **Block Coverage** | 90.8% | | **Instruction Coverage** | 94.0% |   **Таб. 1**. *Таблицата за code coverage summary в code coverage HTML отчета* |  |  |

Code Statistics 13 classes, 339 executable lines

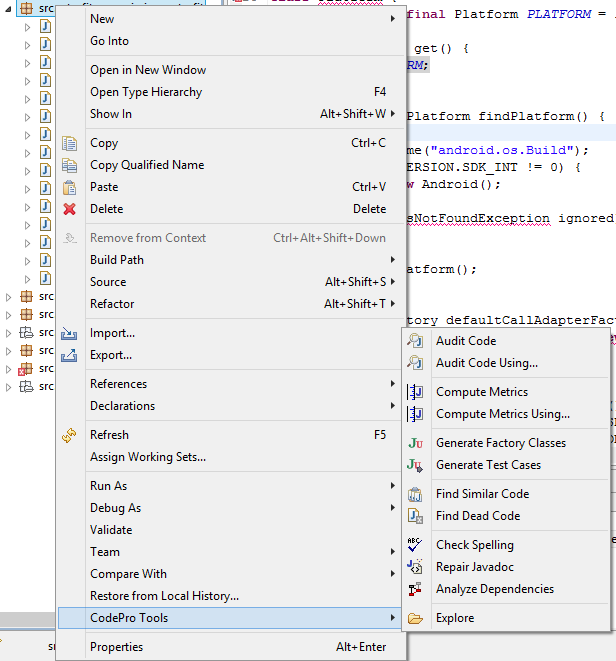
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Total Packages** | 7 | | **Total Files** | 13 | | **Total Classes** | 13 | | **Total Executable Lines** | 339 |   **Таб. 2**. *Таблицата за code statistics в code coverage HTML отчета* |  |  |

# Използване на CodePro Analytix

В тази точка ще демонстрираме основните начини за използване на CodePro Analytix чрез няколко примера.

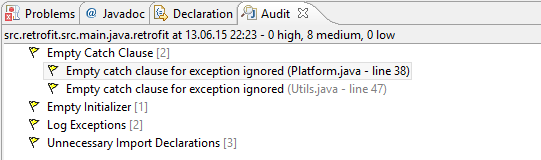
## Auditing

За да използваме Code Auditing трябва да натиснем десен бутон върху някой от пакетите или върху файл и да изберем **Audit Code** от **CodePro** Tools менюто.



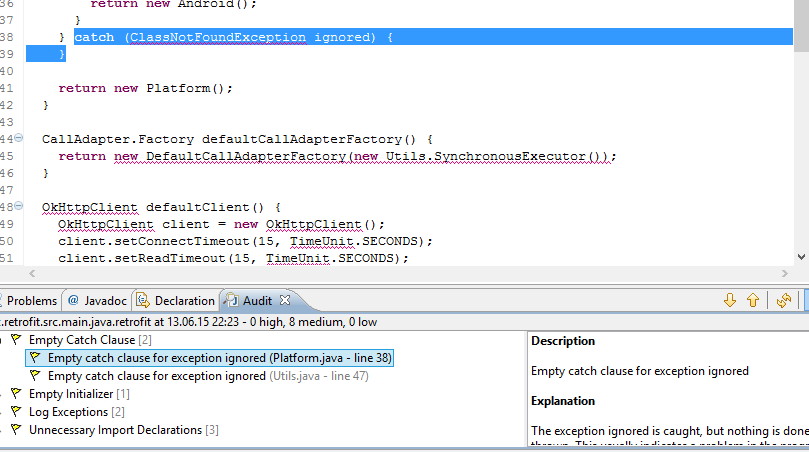
**Фиг. 26**. *Audit Code Example 1*

След като изпълним предната стъпка в таба Audit ще се появят отделните проблеми който са доловени от CodePro Analytix. Те имат три приоритета high, medium и low.



**Фиг. 27**. *Audit Code Example 2*

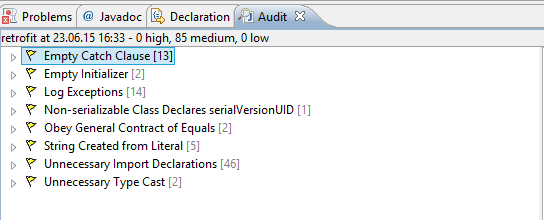
Ако цъкнем два пъти върху даден компонент то директно ще се отвори файла в който присъства проблема и ще го маркира което прави ускорява работата и помага за подобряването на качеството на кода.



**Фиг. 28**. *Audit Code Example 3*

## Прилагане на Audit върху трите тестови проекта и разглеждане на резултата

### Retrofit

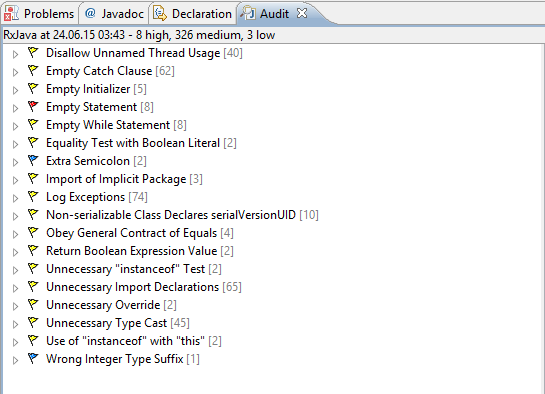


**Фиг. 29**. *Retrofit Audit*

Audit долавя следните проблеми в проекта:

* + Празна клауза за хващане на изключения
  + Празен инициализатор
  + Никъде не се съхранява това, че е възникнало изключение
  + Клас който не може да се сериализира не може да има serialVersionUID
  + Грешен тип на параметъра
  + Конструиране на стринг чрез литерал
  + Включване на пакети които не се използват в съответния файл
  + Кастване към тип при положение, че е ненужно

### Rxjava

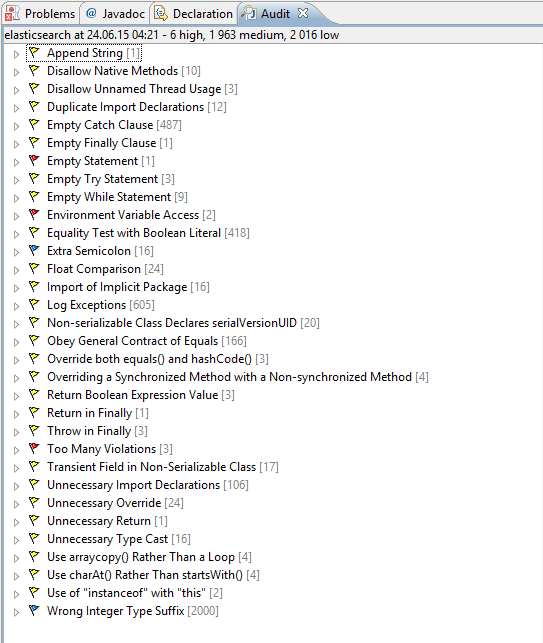


**Фиг. 30**. *RxJava Audit*

Audit долавя следните проблеми в проекта:

* + На нишките не е дадено име (по лесно дебъгване)
  + Празна клауза за хващане на изключения
  + Празен инициализатор
  + Точка и запетая вместо израз или блок
  + Празен while цикъл
  + Равенство с булев оператор
  + Допълнителна точка и запетая
  + Няма нужда от импорт когато пакетът е добавен по подразбиране
  + Никъде не се съхранява това, че е възникнало изключение
  + Клас който не може да се сериализира не може да има serialVersionUID
  + Грешен тип на параметъра
  + Използване на if оператор за връщане на булева стойност
  + Ненужен instanceof тест
  + Включване на пакети които не се използват в съответния файл
  + Предефинираните методи трябва да правят нещо повече от това да извикват предефинирания метод от суперкласа
  + Кастване към тип при положение, че е ненужно
  + This не трябва да бъде тествано с instanceof
  + Long типа трябва да използва L за суфикс

### Elasticsearch



**Фиг. 31**. *ElasticSearch Audit*

Audit долавя следните проблеми в проекта:

* Използването на символ е по-бързо от използването на String
* Да се избягва използването на native методи
* На нишките не е дадено име (по лесно дебъгване)
* Дублиране на import-натите пакети
* Празна клауза за хващане на изключения
  + Празна finally клауза
  + Точка и запетая вместо израз или блок
  + Празна try клауза
  + Празен while цикъл
  + Не използваите System.getenv() за да достъпите променливи на средата
  + Булевите литерали не трябва да бъдат използвани при сравняване
  + Допълнителна точка и запетая
  + Float не трябва да се сравняват с == и != заради проблеми при закръглянето
  + Никъде не се съхранява това, че е възникнало изключение
  + Клас който не може да се сериализира не може да има serialVersionUID
  + Грешен тип на параметъра
  + Ако се предефинира equals() то трябва да се предефинира и hashCode() и обратното
  + Синхронизиран метод не може да бъде предефиниран от не синхронизиран
  + Използване на if оператор за връщане на булева стойност
  + Finally блока не трябва да съдържа return
  + Finally блока не трябва да съдържа throw
  + Класове които не импортват Serializable не трябва да използват transient при декларация
  + Включване на пакети които не се използват в съответния файл
  + Предефинираните методи трябва да правят нещо повече от това да извикват предефинирания метод от суперкласа
  + Void метод не трябва да съдържа return
  + Кастване към тип при положение, че е ненужно
  + Метода arraycopy() може да бъде използван за копиране на масив вместо цикъл
  + По добре да се използва charAt() вместо startsWith() когато се търси символ (по бързо)
  + This не трябва да бъде тествано с instanceof
  + Long типа трябва да използва L за суфикс

# Мнение на авторите

Инструмента е доста полезен за контролиране на качеството на кода и за проследяването на начина на писане на код особено при големи проекти където е трудно да се проследи всичко. Понякога Audit-а хваща грешки които е нямало как да се избегнат поради една или друга причина, но това би трябвало да се случва рядко ако кода се пише качествено. Също така спестява писане при генерирането на Factory класове и Test класове. Откриването на Dead и Similar код е нещо което подобрява качеството на кода значително. Като цяло ценен инструмент който спестява време.

# Библиография

https://developers.google.com/java-dev-tools/codepro/doc/?hl=en