Государственное бюджетное образовательное учреждение среднего профессионального образования Новосибирской области

Новосибирский химико- 630102 г. Новосибирск, ул. Садовая, 26,

технологический колледж. Тел./факс: (383) 266-00-44, тел.: (383) 266-00-54,

nhtk@mail.ru, http://nhtk-edu.ru

им. Д.И. Менделеева

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ**

**ПРАКТИЧЕСКОЙ РАБОТЫ**

Разработчик:

Гуров Д.А

Новосибирск – 2024

**Практическая работа №9 «Использование инструментария анализа качества»**

**Теоретический материал**

В основе модели качества SonarQube лежит реализация методологии SQALE (Software Quality Assessment based on Lifecycle Expectations) с определенными дополнениями.

Как известно, методология SQALE фокусируется в основном на сложности поддержки кода (maintainability) и не учитывает риски проекта. Например, если сегодня в проекте обнаружилась критическая проблема безопасности, строгое следование методологии SQALE обязывает вас устранить все уже существующие проблемы с надежностью (reliability), возможностью изменений (changeability), тестируемостью (testability) и т. д., и только затем вернуться к новой критической проблеме.

На самом деле, если потенциальные проблемы существуют в коде давно и не проявляют себя в виде пользовательских баг-репортов, гораздо важнее сфокусироваться на исправлении новых багов.

С учетом этого, разработчики SonarQube модифицировали модель качества, основанную на SQALE, чтобы акцентировать внимание на следующих важных моментах:

**Модель качества должна быть максимально простой в использовании.**

**Баги и уязвимости не должны теряться среди проблем поддерживаемости (maintainability).**

**Наличие серьезных багов и уязвимостей в проекте должно приводить к тому, что требования Quality Gate не выполнены.**

**Проблемы поддерживаемости кода тоже важны, и их нельзя игнорировать.**

**Вычисление стоимости устранения проблем (использование модели анализа SQALE) важно и должно выполняться.**

Стандартный Quality Gate SonarQube использует следующие значения метрик для определения того, что код успешно прошел проверки:

**0 новых багов.**

**0 новых уязвимостей.**

**Коэффициент технического долга на новом коде <= 5%.**

**Покрытие нового кода не ниже 80%.**

Команда Sonar определила семь смертных грехов разработчиков, увеличивающих технический долг:

**Баги и потенциальные багги.**

**Нарушение стандартов кодирования.**

**Дублирование кода.**

**Недостаточное покрытие модульными тестами.**

**Плохое распределение сложности.**

**Спагетти-дизайн.**

**Недостаточно или слишком много комментариев.**

Платформа SonarQube предназначена для того, чтобы помогать бороться с этими семью грехами.

**Функциональное тестирование**

Функциональное тестирование – это проверка ПО на правильность функционирования в идеальных условиях, в отличие от нефункционального, где либо условия неидеальны (нагрузочное тестирование), либо тестируется не правильность функционирования, а что-то иное (тестирование удобства пользования или структура программы).

Есть много приложений, для которых производительность и удобство пользования некритичны. Во всяком случае, часто требования к ПО содержат только функциональную часть. И практически не бывает требований к ПО без функциональной части. Отсюда делаем вывод, что функциональное тестирование проводить нужно для любого ПО.

Как правило, функциональное и нефункциональное тестирование ПО можно проводить параллельно, поэтому обычно это делается разными людьми или командами. В большинстве источников указывается, что функциональное тестирование – это синоним black-box тестирования (при котором программа рассматривается как черный ящик).

**Black-box, white-box, gray-box тестирование.**

По степени доступа к коду различают два типа тестирования: тестирование «черного ящика» (black-box), или функциональное тестирование – без доступа к коду, и «белого ящика» (white-box), или структурное тестирование – тестирование кода.

В случае «черного ящика» программа рассматривается как конечный автомат, с входными и выходными данными, набором внутренних состояний и переходов между ними. Тестируется правильность поведения программы при различных входных данных и внутреннем состоянии. Правильность определяется исходя главным образом из спецификации, а также любыми другими способами, кроме изучения кода (см. лекцию 1).

В случае «белого ящика» тестировщик пишет тесткейсы, основываясь исключительно на коде программы (тесты на правильность кода).

Расширение black-box тестирования, в котором также применяется изучение кода, называется тестированием «серого ящика» (gray-box). В этом случае правильность поведения определяется любым удобным способом, в том числе изучением кода, что позволяет писать более эффективные black-box тесты (то есть тесты на правильность поведения).

Терминология приведена по книге A Practitioner's guide to Software Test Design (Lee Copeland).

**Методы отбора тестов для black-box тестирования**

Любая программа может рассматриваться как конечный автомат, с входными и выходными данными, набором внутренних состояний и переходов между ними.

Чтобы провести полное тестирование программы, нужно проверить правильность ее поведения при всех возможных комбинациях входных данных и во всех возможных внутренних состояниях. Это невозможно из-за огромного числа комбинаций даже в простейших случаях. Поэтому на практике отбираются только наиболее важные тесты, такой отбор можно производить несколькими методами.

Тестирование сценариев использования – юз-кейсов (use-cases)

Чтобы удостовериться в правильности перехода программы между различными внутренними состояниями, в идеале следует протестировать все возможные переходы между каждым из состояний (не только одношаговые, но и многошаговые, то есть все пути в графе состояний).

Чтобы уменьшить число тестов, можно проверить только те переходы, которые имеют смысл для пользователя. Use-case – это логически завершенная последовательность действий. Например, открытие файла в Notepad – это use-case, а выбор пункта меню «Открыть файл» в Notepad – это не use-case, а лишь первый шаг юзкейса «открытие файла».

Тестирование сценариев является самым необходимым видом тестирования. Программа должна выполнять операции, для которых она предназначена. Если пользователь может выбрать пункт меню, но файл не открывается – это очень серьезный баг.

Здесь проверяется правильность перехода программы между внутренними состояниями при выполнении определенных операций (т. е. при определенных входных данных).

**Тестирование классов эквивалентности**

Чтобы удостовериться в правильности поведения программы при различных входных данных, в идеале следует протестировать все возможные значения для каждого элемента этих данных (а также все возможные сочетания входных параметров).

Например, пусть мы тестируем программу для отдела кадров, в ней есть поле «Возраст соискателя».

Пример взят из книги A Practitioner's guide to Sofware Test Design (Lee Copeland).

Требования по возрасту у нас будут такие:

0–13 лет – не нанимать;

14–17 лет – можно нанимать на неполный день; 18–54 года – можно нанимать на полный день; 55–99 лет – не нанимать.

Чтобы проверить все возможные разрешенные данные (только разрешенные!) нам нужно протестировать ввод чисел от 0 до 99. (Возможен ведь еще ввод отрицательных чисел и нечисловых данных.) Так ли необходимо тестировать все числа от 0 до 99? В случае, если программа анализирует каждое чиcло по отдельности, вот таким образом, то видимо, да:

...

if (age == 13) hireStatus=«NO»;

if (age == 14) hireStatus=«PART»; if (age == 15) hireStatus=«PART»; if (age == 16) hireStatus=«PART»; if (age == 17) hireStatus=«PART»; if (age == 18) hireStatus=«FULL»;

...

Но к счастью, программы обычно пишут по-другому:

if (age >= 0 && age <=13)

hireStatus=«NO»;

if (age >= 14 && age <=17) hireStatus=«PART»;

if (age >= 18 && age <=54) hireStatus=«FULL»;

if (age >= 55 && age <=99) hireStatus=«NO»;

Становится очевидным, что можно протестировать одно из чисел каждого диапазона. Например: 5, 15, 20, 60. А также граничные значения (первое и последнее значения из каждого диапазона): 0, 13, 14, 17, 18, 54, 55, 99.

Чтобы уменьшить количество тестируемых значений, производится:

а) разбиение множества всех значений входной переменной на подмножества (классы эквивалентности), а затем;

б) тестирование одного любого значения из каждого класса. Все значения из каждого подмножества должны быть эквива-

лентны для наших тестов. То есть, если тест проходит успешно для одного значения из класса эквивалентности, он должен проходить успешно для всех остальных. И наоборот, если тест не проходит для одного значения, он не должен проходить для всех остальных.

В данном случае имеем 12 классов эквивалентности (каждое из 8 граничных значений по сути является отдельным классом).

Чтобы проверить правильность работы программы на всех разрешенных данных, нужно провести 12 тестов.

Запрещенные данные тестируются аналогично – можно выделить классы эквивалентности «дробное число от 0 до 99», «отрицательное число», «число больше 99», «набор букв», «пустая строка» и т. д.

Таким образом, метод классов эквивалентности можно разделить на три этапа:

1. Тестирование разрешенных значений.
2. Тестирование граничных значений.
3. Тестирование запрещенных значений.

Часто в литературе второй и третий этапы называют отдельными методами, но сути это не меняет.

**Попарное тестирование**

Метод классов эквивалентности применяется для тестирования каждого входного параметра по отдельности.

Пусть наша программа принимает на вход десяток параметров. Баги, возникающие при определенном сочетании всех десяти параметров, довольно редки. Вообще, взаимное влияние параметров, о котором пользователь не знает – это баг интерфейса (интерфейс интуитивно не понятен).

Чаще всего будут встречаться ситуации, в которых один параметр влияет на один из оставшихся, т. е. самыми частыми будут баги, возникающие при определенном сочетании двух каких-то параметров.

Таким образом, можно упростить себе задачу и протестировать все возможные значения для каждой из пар параметров. Такой подход называется попарным тестированием (pairwise testing).

Вот пример. Пусть имеется 3 двоичных входных параметра (3 чекбокса). Количество всех возможных комбинаций – 2 в степени 3 = 8 , значит, нужно произвести 8 тестов. Давайте попробуем сэкономить, тестируя чекбоксы попарно.

Выпишем все комбинации для первого и второго чекбоксов: 1-й 2-й

|  |  |
| --- | --- |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

Добавим третий столбец так, чтобы во втором и третьем столбце получились все 4 двоичные комбинации. Это можно сделать разными способами, мы сделаем так (на первый столбец можно не обращать внимания):

1-й 2-й 3-й

0 0 0

0 1 0

1 0 1

1 1 1

Итак, с помощью четырех наборов входных данных (четырех тестов) мы протестируем две пары параметров: первый со вторым и второй с третьим. Осталось протестировать пару «первый с третьим».

Выпишем отдельно 1 и 3 столбцы:

|  |  |
| --- | --- |
| 1-й | 3-й |
| 0 | 0 |
| 0 | 0 |
| 1 | 1 |
| 1 | 1 |

Как видно, мы имеем здесь две из четырех возможных комбинаций. Комбинации «01» и «10» здесь отсутствуют, а комбинации

«00» и «11» присутствуют два раза. Ну что же, добавим еще 2 строки (еще два теста)

|  |  |
| --- | --- |
| 1-й | 3-й |
| 0 | 0 |
| 0 | 0 |
| 1 | 1 |
| 1 | 1 |
| 0 | 1 |
| 1 | 0 |

Вернем второй столбец на его законное место: 1-й 2-й 3-й

0 0 0

0 1 0

1 0 1

1 1 1

0 1

1 0

Выходит, что последние два теста можно проходить при любых значениях второго параметра. Можно дописать для определенности нули в эти пустые места:

1-й 2-й 3-й

0 0 0

0 1 0

1 0 1

1 1 1

0 0 1

1 0 0

Получаем 6 тестов вместо 8 при полном переборе. Можно ли сэкономить еще? Оказывается, можно. Вернемся к 1 шагу:

1-й 2-й

|  |  |
| --- | --- |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

Давайте допишем третий столбец другим способом, поменяв порядок комбинаций:

1-й 2-й 3-й 0 0 1

0 1 0

1 0 0

1 1 1

Все комбинации для 1 и 2, а также для 2 и 3 параметра здесь есть. Отлично.

Посмотрим теперь на комбинации 1 и 3 параметра

1-й 3-й

0 1

0 0

1 0

1 1

Ого! Что мы видим? Изменив порядок значений в третьем столбце, мы одним махом убили двух зайцев: скомбинировали и 2-й с 3-м, и 1-й с 3-м параметры.

Итого имеем всего 4 строки, то есть 4 теста, эквивалентные первоначальным шести:

1-й 2-й 3-й 0 0 1

0 1 0

1 0 0

1 1 1

Полный перебор всех комбинаций в третьем столбце гарантированно даст минимальное количество тестов. Однако, судя по тому, что алгоритмы такой минимизации разрабатываются до сих пор, полный перебор неприемлем из-за большого времени исполнения. Существуют программы, дающие приемлемый результат в приемлемое время, например, программа PICT от Microsoft.

Ортогональный массив OA(N,k,s,t) – это двумерный массив из N рядов (итераций) и k колонок (факторов) из набора S (т. е. факторы могут принимать любое из s значений), обладающий свойством: выбрав любые t колонок (0<=t<=k) мы получим в рядах все комбинации сочетаний из s по t (Количество повторений одинаковых комбинаций обозначают через λ. Чаще всего рассматривают массивы, где λ = 1, т. е. каждая комбинация встречается только один раз). Параметр t называют мощностью ортогонального массива.

В попарном тестировании применяется ортогональный массив мощности 2 – это двумерный массив такой, что любые 2 колонки этого массива содержат все возможные комбинации (пары) значений, хранящихся в массиве.

Использование информации о программе при gray-box тестировании.

Главным минусом black-box тестирования является то, что тестировщик не знает, какую часть ПО он тестирует. Некоторые существующие пути в программе (о которых нет информации ни в требованиях, ни в документации), могут никогда не быть проверены. Уменьшить количество таких путей можно путем анализа внутреннего устройства программы.

Информация о базе данных

Если программа использует для своей работы какую-либо базу данных, мы можем проанализировать типы полей, в которые записываются переменные программы. А потом проанализировать ограничения, которые накладывает база данных.

Например, если вводимая фамилия пользователя записывается в поле типа «строка» длиной 128 символов, мы должны:

попробовать найти фамилию длиннее, чем 128 символов – здесь будет довольно серьезный баг, если такие фамилии существуют – человек с такой фамилией не сможет воспользоваться нашей системой;

вне зависимости от того, существуют или нет такие фамилии, попробовать ввести строку длиннее 128 символов – программа не должна ломаться (должно показываться внятное сообщение об ошибке).

Информация о других внешних системах

Если программа интегрируется с другими внешними системами, помимо базы данных, можно также проанализировать ограничения таких систем. Например, если мы тестируем почтовый IMAPклиент, следует убедиться, что он корректно обрабатывает длинные пути к папкам на сервере (чаще всего, ограничение на длину пути составляет 255 символов)

Информация о коде программы

Если мы имеем доступ к коду программы, мы можем

а) увидеть специальные случаи, которые не попали в документ с требованиями и которые необходимо протестировать или, напротив;

б) увидеть, что какие-то вещи тестировать не имеет смысла.

**Методы отбора тестов для White-Box тестирования**

White-box тестирование является не функциональным, а структурным – тестируется код программы. От тестировщика требуются, таким образом, навыки программиста. Главным минусом такого тестирования является то, что тестировщик никогда не сможет обнаружить, что какая-то функциональность не реализована, поскольку нельзя протестировать код, который не написан.

Различают два метода отбора тестов:

**Тестирование путей исполнения (control-flow testing).**

**Тестирование работы с данными (data-flow testing).**

Подробно эти методы описаны в книге A Practitioner's guide to Sofware Test Design (Lee Copeland). Описание их выходит за рамки данной лекции, поскольку они не являются методами функционального тестирования.

**Тестирование безопасности**

Тестирование безопасности – процесс достижения совершенства в открытом космосе при условии, что с Вашим скафандром что-то не так. Но если вернуться на Землю, то тестирование безопасности веб-приложения – это попытка найти все те места, в которых могли допустить ошибку разработчики или просто не предусмотреть / забыть (подчеркните ваш случай).

Веб-приложение и веб-сервер неразрывно связаны. Тестирование одного без другого не даст полной картины бедствия. Тестируя защиту веб-приложения, мы ищем уязвимые места для атаки на пользователей. Тестируя защиту веб-сервера, мы ищем уязвимые места для атаки на сервер, его инфраструктуру.

Нужно все это в первую очередь людям: простым смертным, решившим заказать пиццу, а не отправить данные своей карты непонятно куда; простым владельцам пиццерий, не беспокоящимся, что кто-то смог бесплатно научиться заказывать пиццу через их приложение; простым разработчикам, которым не придется править код в 3 часа утра.

Как правило, небольшие организации, имеющие свой сайт и небольшой сервер с битриксом, думают, что они слишком малы для того, чтобы стать мишенью для атаки. И в этом они заблуждаются. Безусловно, вероятность таргетированной атаки на них ниже, чем на финансовых гигантов, но все-таки не равна нулю. В нынешнее время нейронных сетей и повальной автоматизации никто не будет выяснять, большой ли денежный оборот у фирмы. Главное – это количество уникальных посетителей, ведь суммарно в их карманах может оказаться больше «фишек», чем компания получает за год.

Существуют компании, которые уже взломали, и компании, которые еще не взломали. На практике это вопрос времени. Правильные компании, которые тратят на безопасность значительную часть прибыли, – это нормальный порядок вещей для всего цивилизованного мира. В нашей стране, к сожалению, пока к этому не пришли, считая, что ~~«хата наша с краю»~~ студент-старшекурсник вполне способен настроить одну «циску» и запретить доступ к внутренним ресурсам извне (мол, «денег нет, но вы держитесь»). Ведь мы не привыкли тратить деньги на свою безопасность: бюджет отделов по информационной безопасности (ИБ) обычных компаний только сокращается. С другой стороны, многие адекватные компании (читай, иностранные) формируют собственный штат по информационной безопасности, другие многие нанимают специалистов на аутсорс, немногие запускают программы баг-баунти, где любой желающий может принять участие в поиске уязвимостей. Безусловно, все они хотят сохранить свои доходы и свою репутацию.

На что и на кого ориентировано тестирование безопасности?

Кто получит выгоду от этого в первую очередь?

Естественно, в первую очередь от безопасного серфа сайта выигрывает пользователь. Если нет нужды беспокоиться о том, что Ваши персональные данные могут утечь в сеть, то и доверять ресурсу Вы будете больше. А если счастлив пользователь, то счастлив и владелец веб-ресурса (и тем более он счастлив, чем меньше у него рисков потерять финансы).

**ИЗНУТРИ**

Исследование безопасности веб-ресурса – сложная и кропотливая работа, требующая внимательности, фантазии и творческого подхода. Исследователю безопасности необходимо глубокое понимание технической изнанки работы веб-приложения и веб-сервера. Каждый новый проект дает пищу для фантазии, каждый новый инструмент – просторы для творчества. Да и вообще, тестирование безопасности больше похоже на исследовательскую работу – это постоянный поиск и анализ.

Каждый новый проект требует применения новых инструментов, изучения новых технологий, поглощения множества книг и статей, которые достаточно трудно найти. Большая часть информации находится в англоязычном пространстве, что добавляет определенные трудности в освоении материала людям, языком английским не владеющим.

Если говорить о процессе тестирования безопасности, то в целом он не сильно отличается от тестирования обычного: поиск, локализация, воспроизведение, заведение, отчет. Приоритеты, безусловно, зависят от заказчика, от целей тестирования.

К сожалению, некоторые курсы по тестированию безопасности

/ анализу защищенности / аудиту безопасности в интернете внушают, что достаточно пройтись по веб-приложению каким-нибудь сканером безопасности – и все готово! Отчет есть, уязвимости – вот они! Что же еще вам нужно? Но не стоит быть таким наивным. Большинство уязвимостей ищется и находится именно вручную, при внимательном изучении. Они могут быть совершенно несложными, но автоматические сканеры пока еще не способны их обнаружить.

Классификацией векторов атак и уязвимостей занимается сообщество OWASP (Open Web Application Security Project) – международная некоммерческая организация, сосредоточенная на анализе и улучшении безопасности программного обеспечения.

OWASP (http[s://www.o](http://www.owasp.org/))wasp[.org/)](http://www.owasp.org/)) составил список из 10-и самых опасных уязвимостей, которым могут быть подвержены интернет-ресурсы. Сообщество обновляет и пересматривает этот список раз в три года, поэтому он содержит актуальную информацию. Последнее обновление было сделано 2017:

**внедрение кода;**

**некорректная аутентификация и управление сессией;**

**утечка чувствительных данных;**

**внедрение внешних XML– сущностей (XXE);**

**нарушение контроля доступа;**

**небезопасная конфигурация;**

**межсайтовый скриптинг;**

**небезопасная десериализация;**

**использование компонентов с известными уязвимостями;**

**отсутствие журналирования и мониторинга.**

Организация OWASP дополнительно к своему списку из 10 самых опасных уязвимостей разработала методические рекомендации (практикум) по тестированию безопасности веб-приложений. В них подробно, шаг за шагом, описано, как и что необходимо тестировать, на что обратить внимание в первую очередь, а на что – во вторую. Методика носит рекомендательный характер и, конечно, никого ни к чему не обязывает (инженер, проводящий тестирование, некоторые моменты может перенести, а другие – вообще опустить), но, тем не менее, позволяет сделать максимальное покрытие для веб-приложения.

Итак, весь процесс тестирования состоит из двух этапов:

1. Пассивный, во время которого тестировщик пытается понять логику приложения и «играет» с ним. Могут использоваться инструменты для сбора информации. Например, с помощью HTTP прокси можно изучить все HTTP-запросы и ответы. В конце данного этапа тестировщик должен понимать все точки входа приложения (HTTP-заголовки, параметры, куки и пр.).

2. Активный. Во время данного этапа тестировщик проводит тесты в соответствии с методологией. Все тесты разбиты на одиннадцать подразделов:

**сбор информации;**

**тестирование конфигурации;**

**тестирование политики пользовательской безопасности;**

**тестирование аутентификации;**

**тестирование авторизации;**

**тестирование управления сессией;**

**тестирование обработки пользовательского ввода;**

**обработка ошибок;**

**криптография;**

**тестирование бизнес-логики;**

**тестирование уязвимостей на стороне пользователя.**

Какие инструменты используются для анализа безопасности? Оценив объемы бедствия, следует рассмотреть существующие инструменты. Конечно же, главные Ваши аргументы против несправедливости в сетевых именах – это глаза и мозг. На самом же деле, добрые люди разработали огромный инструментарий – начиная от специализированных скриптов, «заточенных» для какой-то одной конкретной цели, и заканчивая целыми комбайнами – готовыми выжать максимальные выводы из минимума вводных. К сожалению, часто такой результат оказывается лже-срабатыванием. Как правило, инженер по тестированию при выборе инструментов основывается на приоритетах: что важнее – время или область покрытия? Современные разработчики довели автоматизацию до небывалых высот, поэтому можно смело следовать принципу Парето: 80% работы скармливать автоматизированным анализаторам, а все

оставшееся «проходить руками». Но, честно говоря, результаты автоматизированных средств все равно придется изучать и проверять.

Приведем небольшой список категорий инструментов:

**сканеры веб-уязвимостей;**

**инструменты для эксплуатации уязвимостей;**

**инструменты криминалистики;**

**сканеры портов;**

**инструменты мониторинга трафика;**

**отладчики;**

**руткит детекторы;**

**инструменты шифрования;**

**инструменты для брутфорса.**

Из-за разгильдяйства разработчиков. Из-за невнимательности.

Из-за халтуры. Из-за лени. Из-за-за…

Но чаще всего уязвимости возникают из-за неопытности. Не все разработчики представляют себе, как злоумышленник будет атаковать их продукт. Некоторые считают, что достаточно экранировать кавычку в пользовательском вводе или спастись

«magic\_quotes» – и можно будет избежать SQL-инъекции. Отсюда и получается, что превентивные меры мы принимаем, а что делать дальше – не знаем. И WAF’ы нас не спасут.

**Порядок выполнения практической работы**

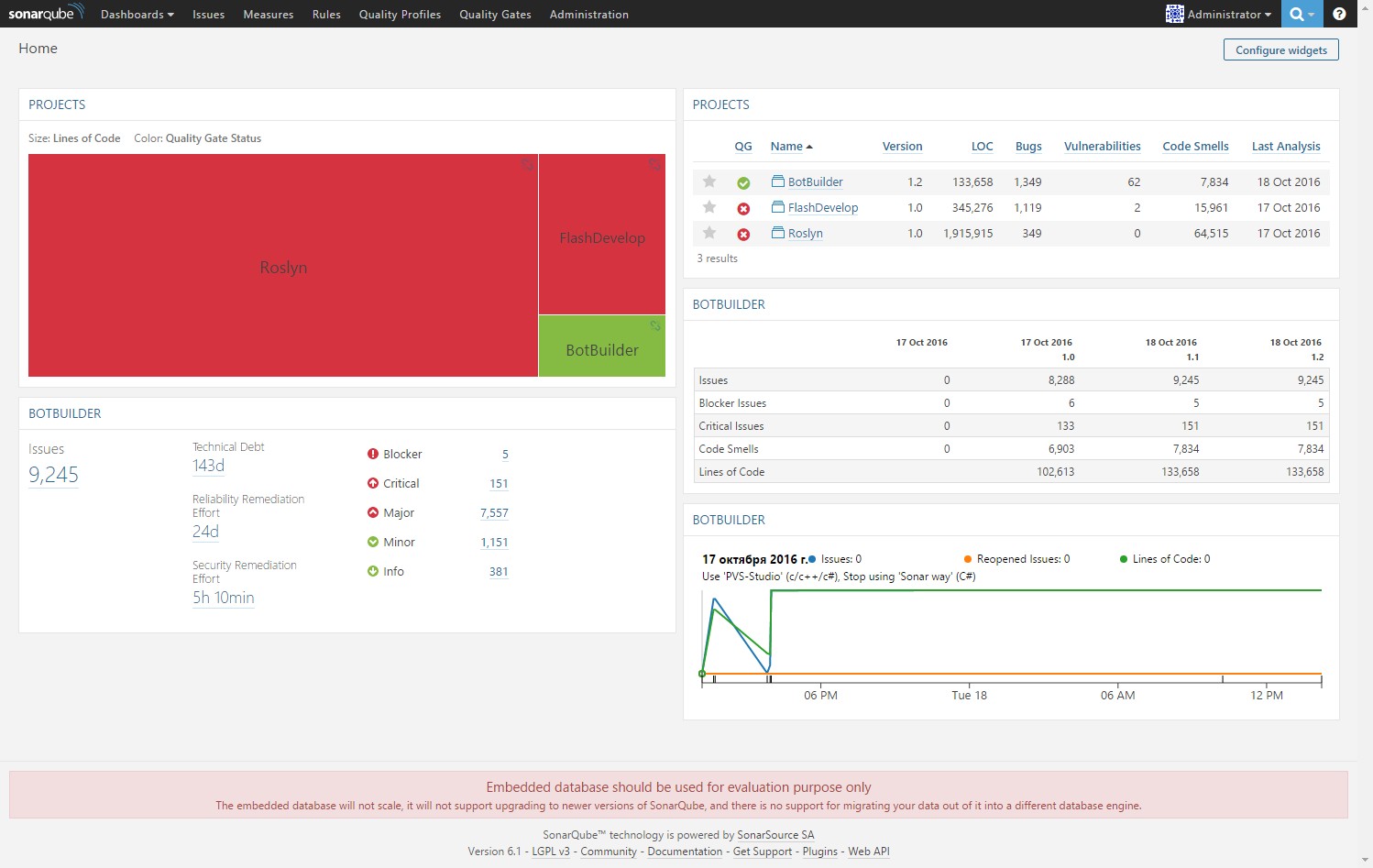
1. Практическая работа выполняется в ***отдельном*** создаваемом студентом ***документе***!
2. Титульный лист с указанием наименования организации, ФИО студента и номер группы
3. Выполнить задания, в отчете написать номер задания, добавить скриншот, описать результат.
4. Ответить на контрольные вопросы
5. Вывод

**Практическое задание**

Рассмотрим более подробно основные возможности SonarQube.

* Главная страница

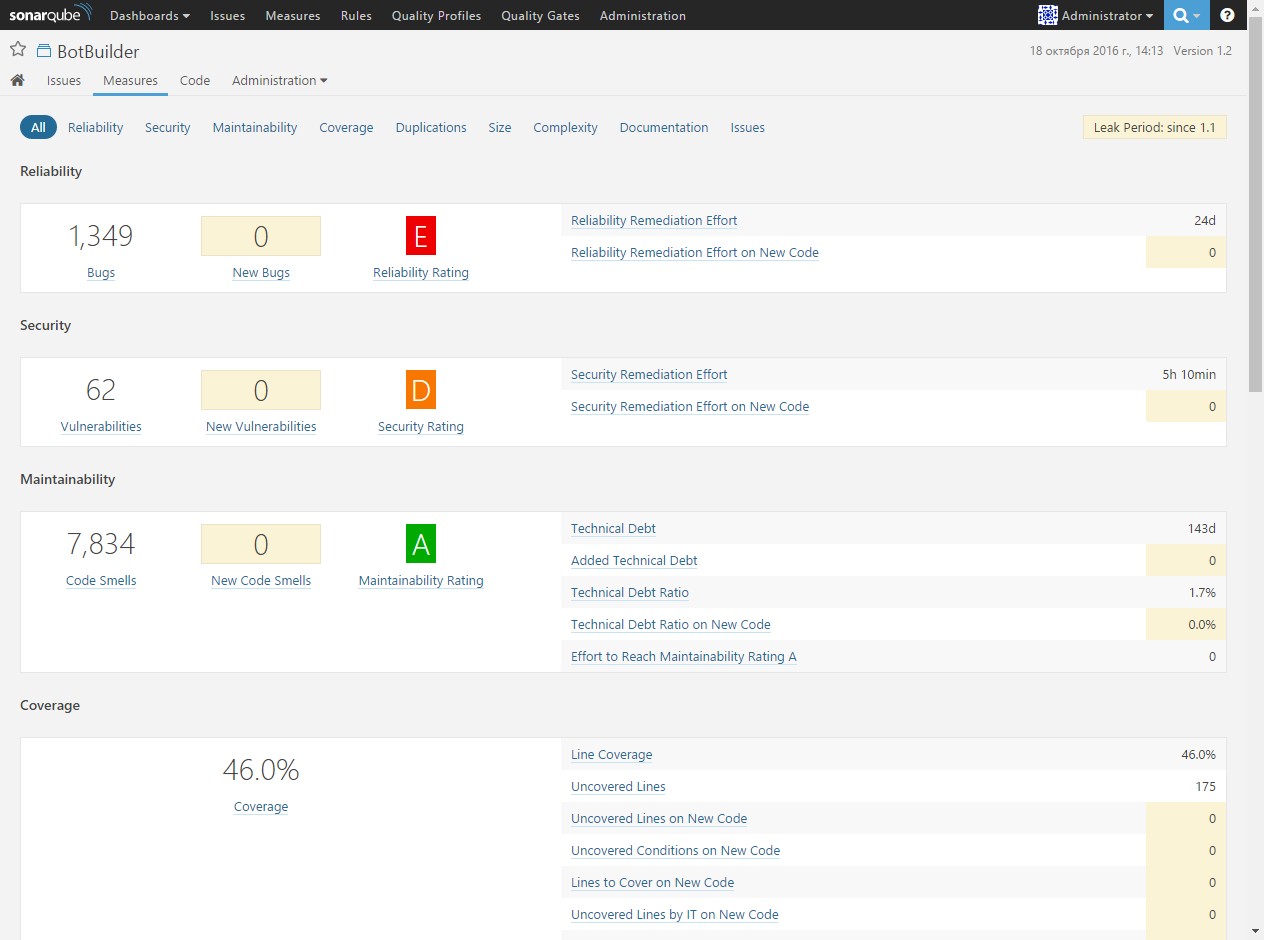
На главной странице SonarQube вы видите список проектов, добавленных в систему, с краткой статистикой по каждому проекту: версия сборки, количество строк кода, количество багов, уязвимостей и признаков «кода с душком», дата последнего анализа:



Содержимое главной страницы можно настроить под ваши цели с помощью большого набора встроенных виджетов, позволяющих визуализировать состояние кода проектов в SonarQube.

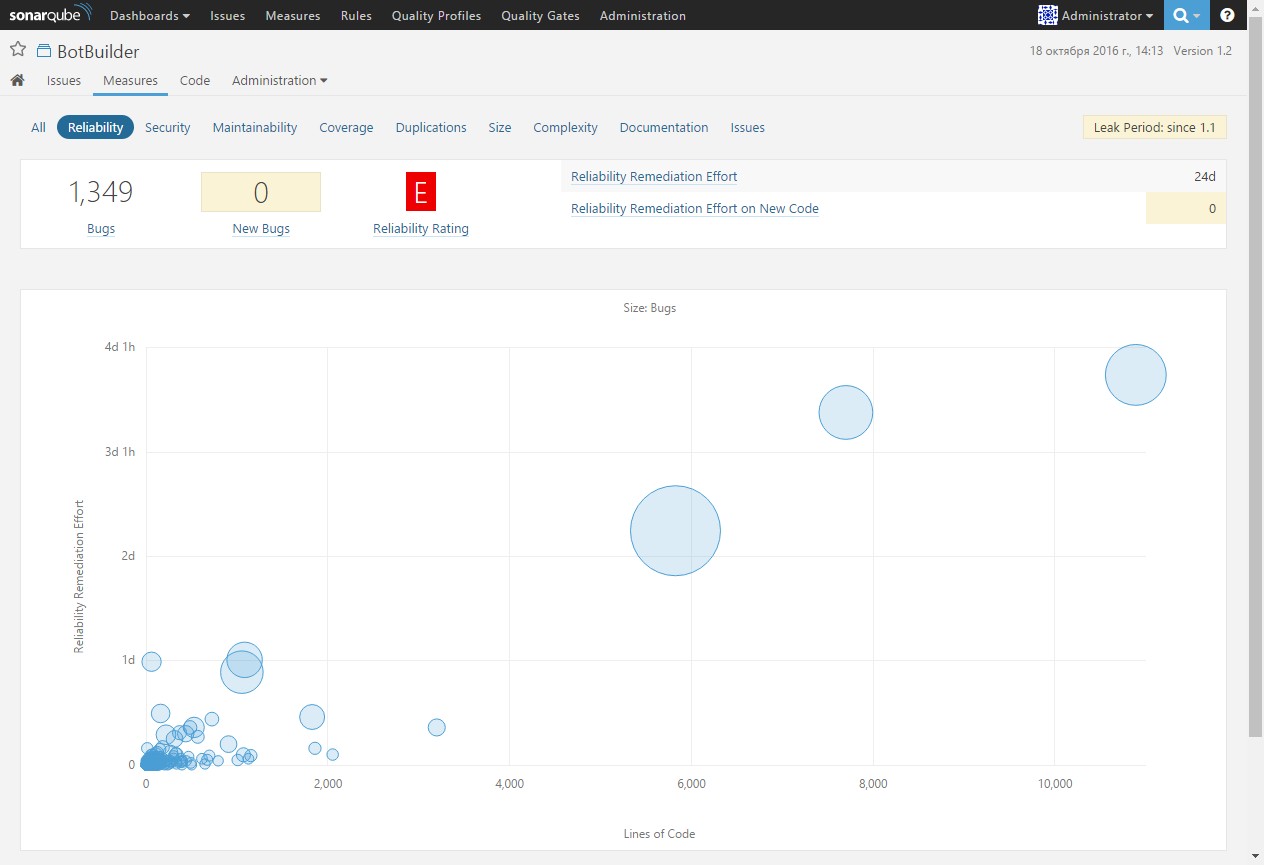
Метрики проекта

Для получения более детальной информации о состоянии проекта перейдем на страницу метрик проекта:

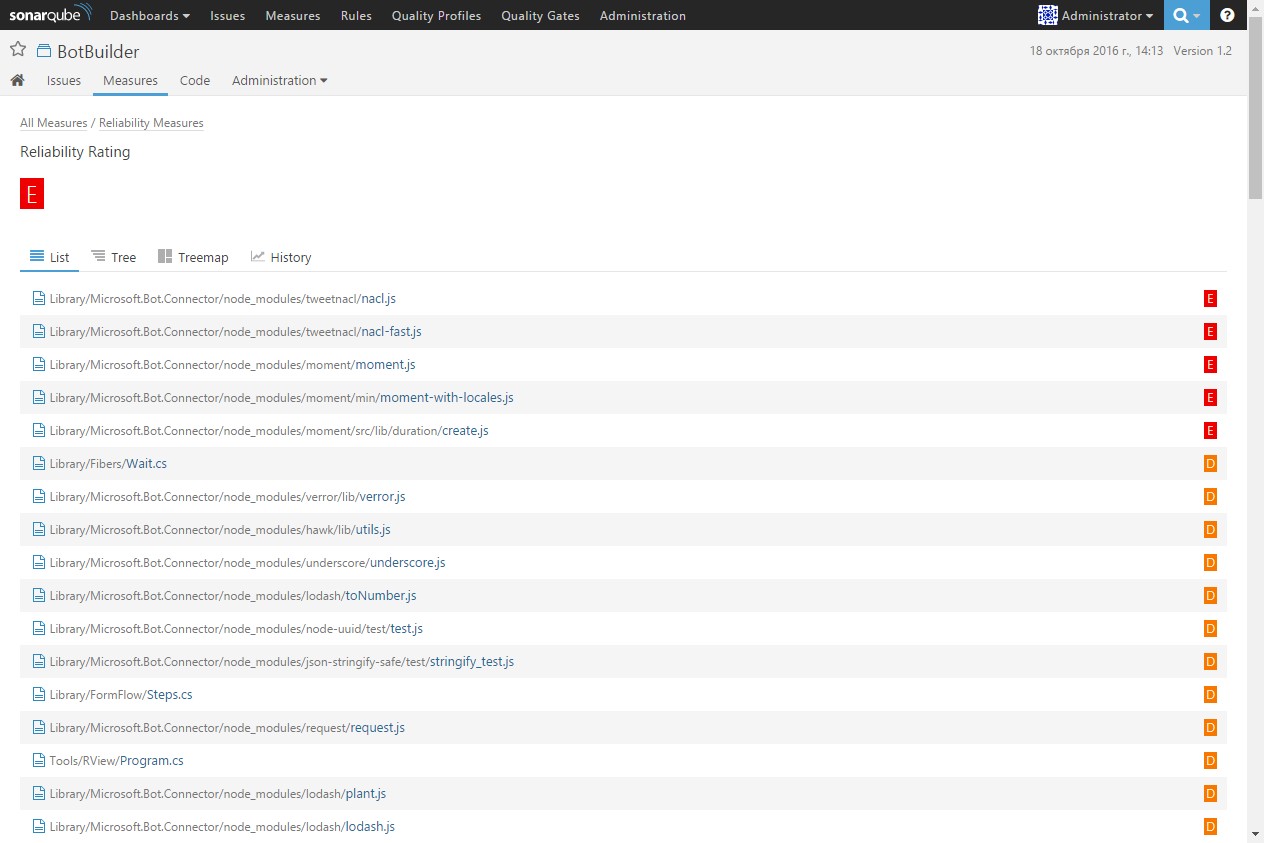


Здесь представлена информация о следующих метриках кода: Reliability (Надежность), Security (Безопасность), Maintainability (Поддерживаемость), Coverage (Покрытие тестами), Duplications (Дублирование), Size (Размер кодовой базы), Complexity (Цикломатическая сложность), Documentation (Документирование кода) и Issues (Ошибки).

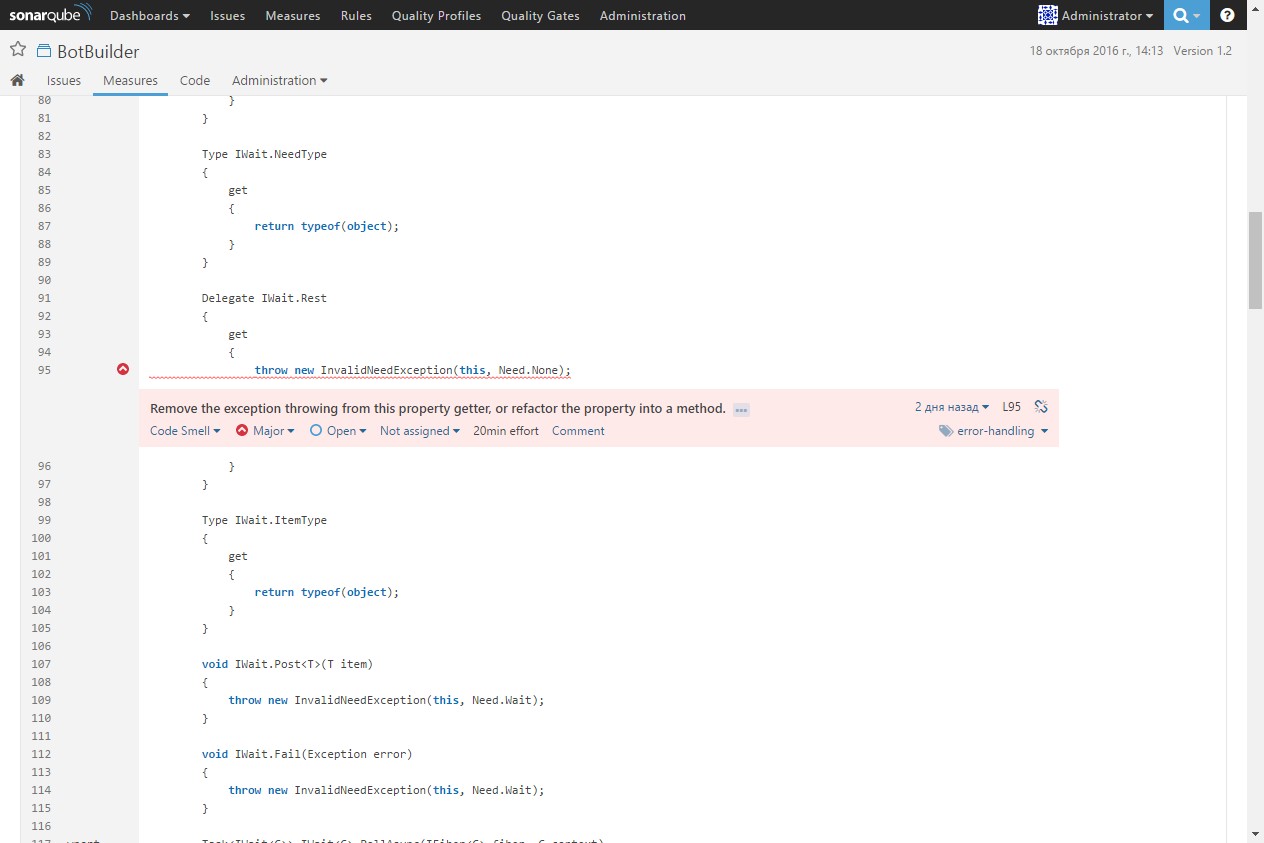
Перейдя к метрике Reliability, мы получаем информацию об общем количестве обнаруженных багов и новые баги, обнаруженные во время последнего анализа, рейтинг надежности кода по шкале от A до E, где E – наихудший рейтинг, свидетельствующий о том, что был найден по крайней мере один blocker баг, а также время, необходимое на устранение всех найденных ошибок:



Платформа SonarQube позволяет анализировать метрики кода сверху вниз, от уровня проекта в целом до отдельных модулей и файлов. Так, например, если вы кликните на рейтинг надежности (Reliability Rating), вы увидите список файлов проекта, отсортированных по возрастанию рейтинга надежности. Это позволит сфокусироваться на наиболее проблемных участках кода:

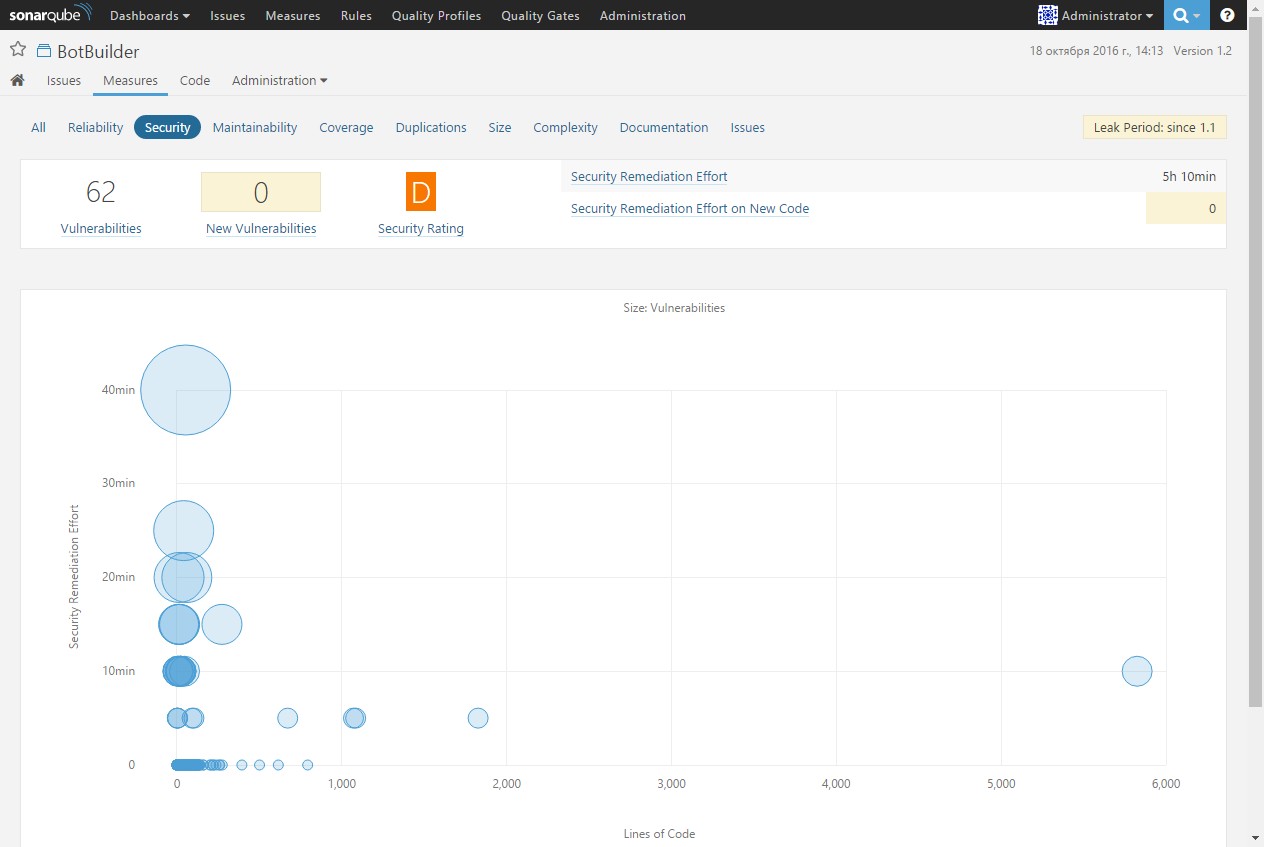


Затем вы можете перейти к файлу с исходным кодом и к конкретным участкам кода, в которых обнаружены ошибки:

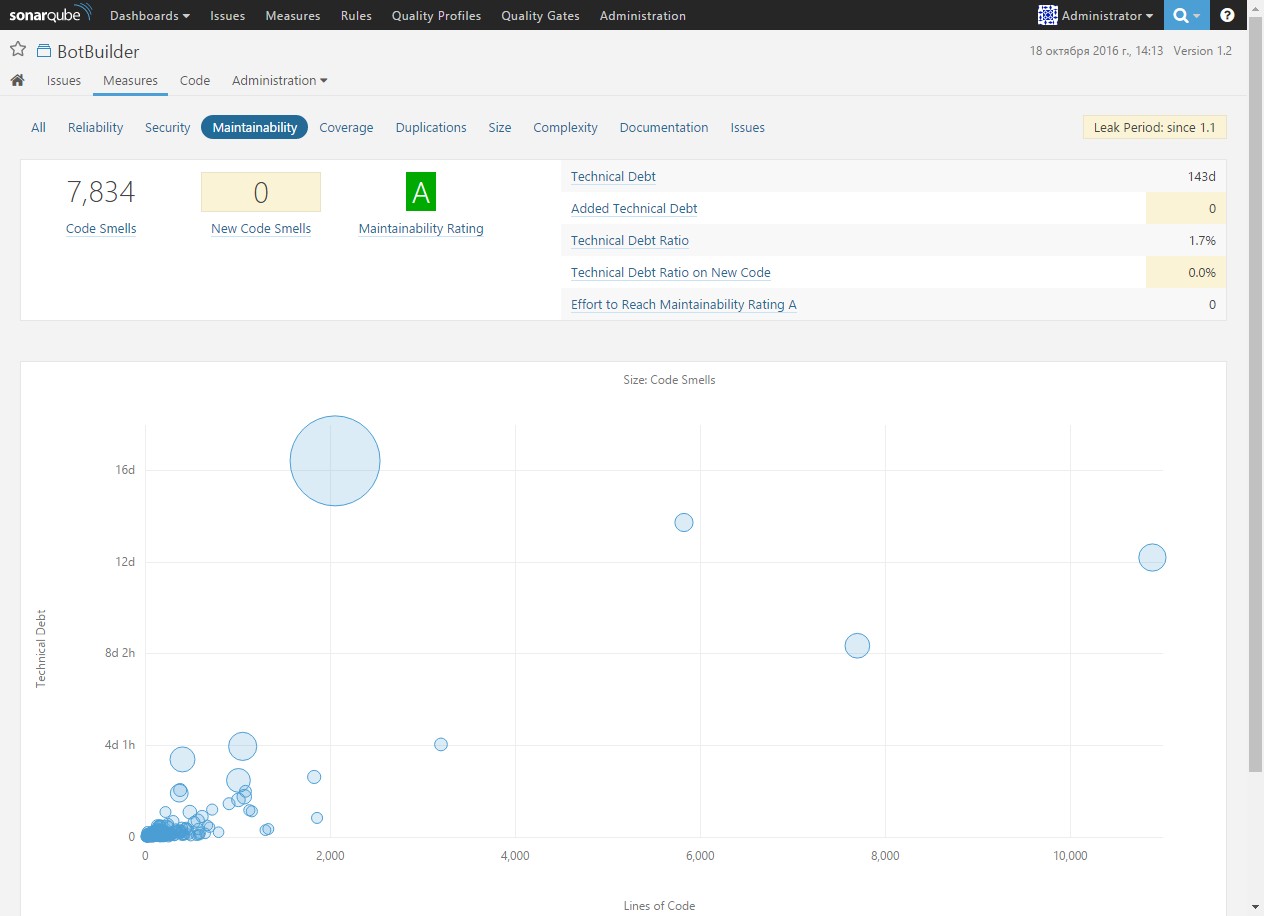


Такая навигация сверху вниз доступна и для других метрик.

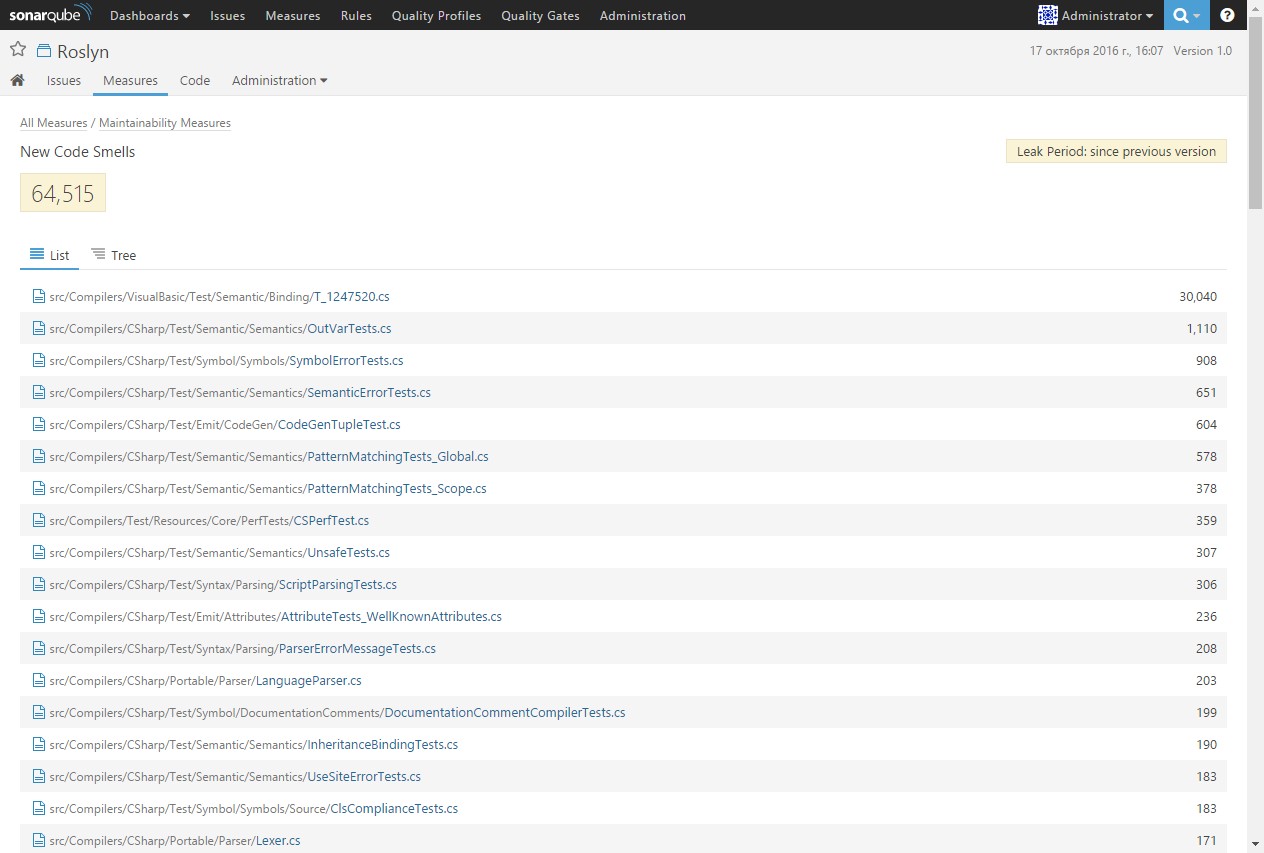
На странице метрики Security доступна информация об общем количестве уязвимостей, новых уязвимостях, рейтинге безопасности (также по шкале от A до E), и времени, которое потребуется на устранение уязвимостей:



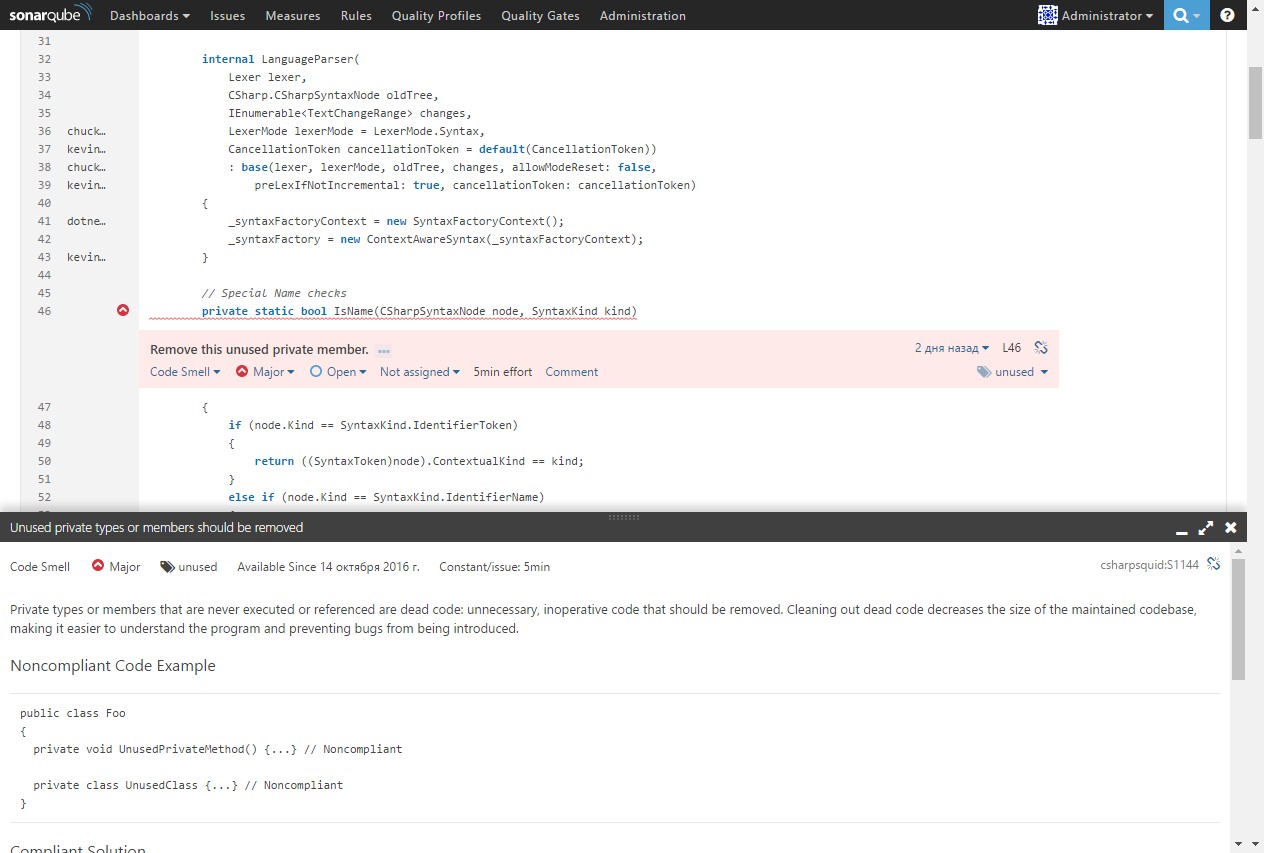
Страница Maintainability содержит информацию о техническом долге в проекте:



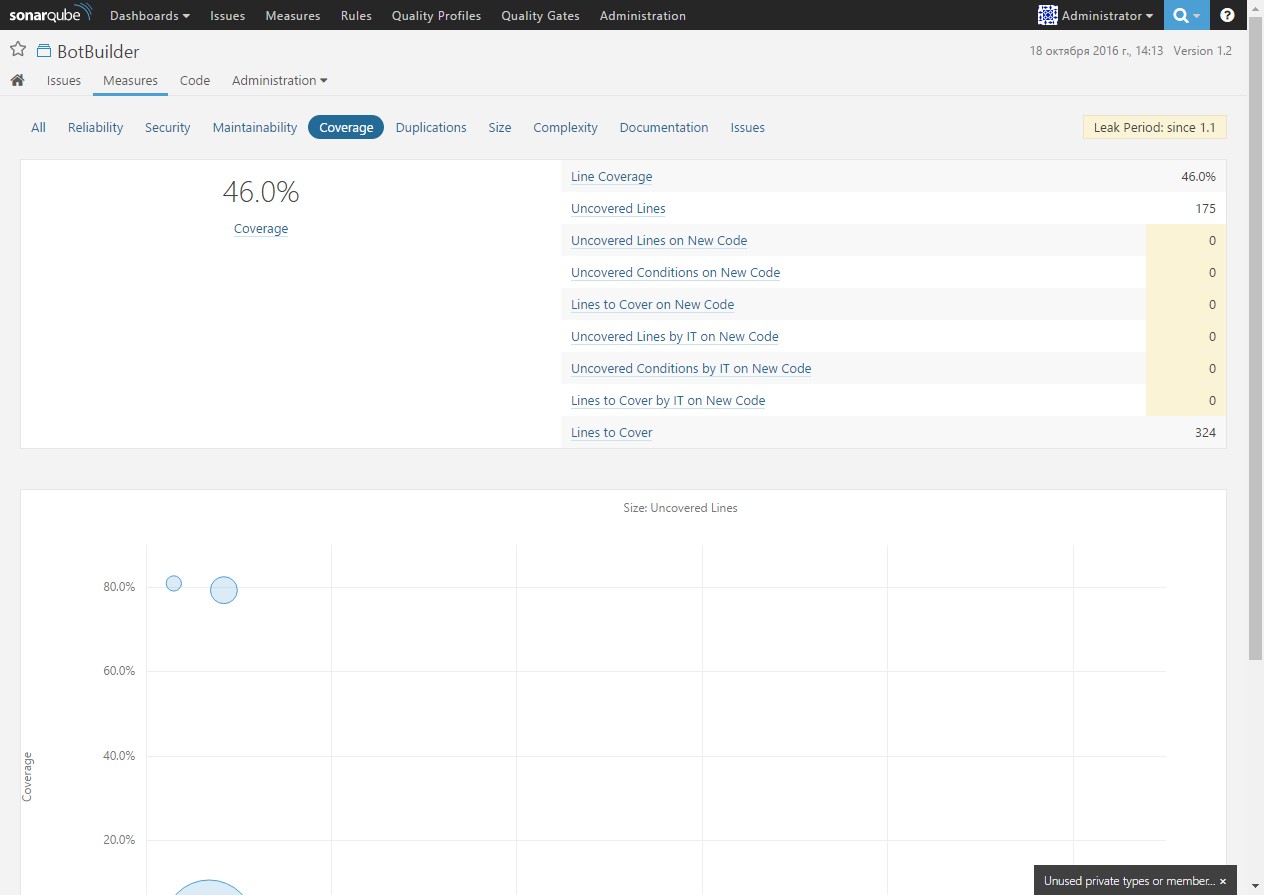
Благодаря навигации «сверху вниз» вы можете перейти к списку файлов, отсортированных по количеству обнаружений кода с душком:



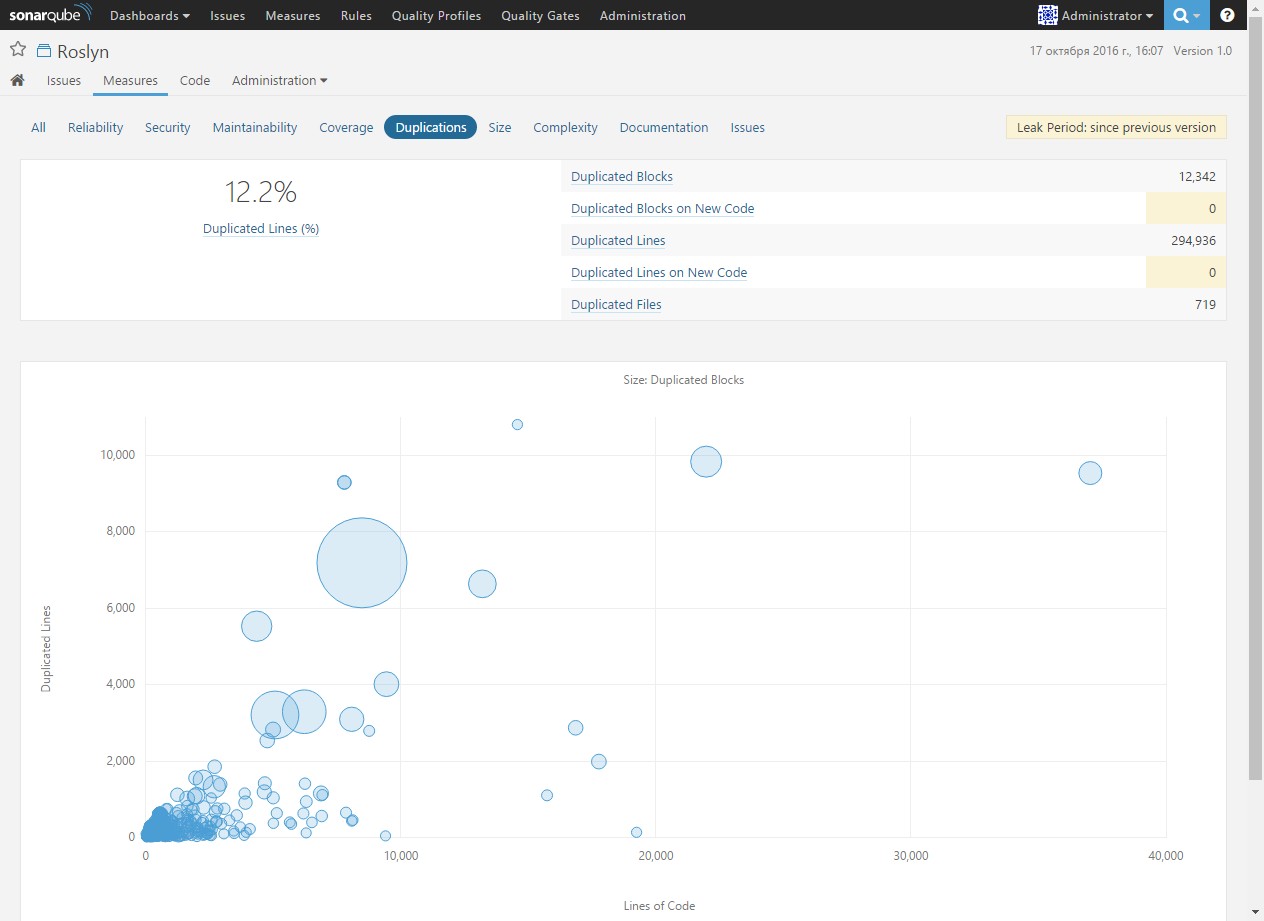
и затем непосредственно к коду, который требует внимания:



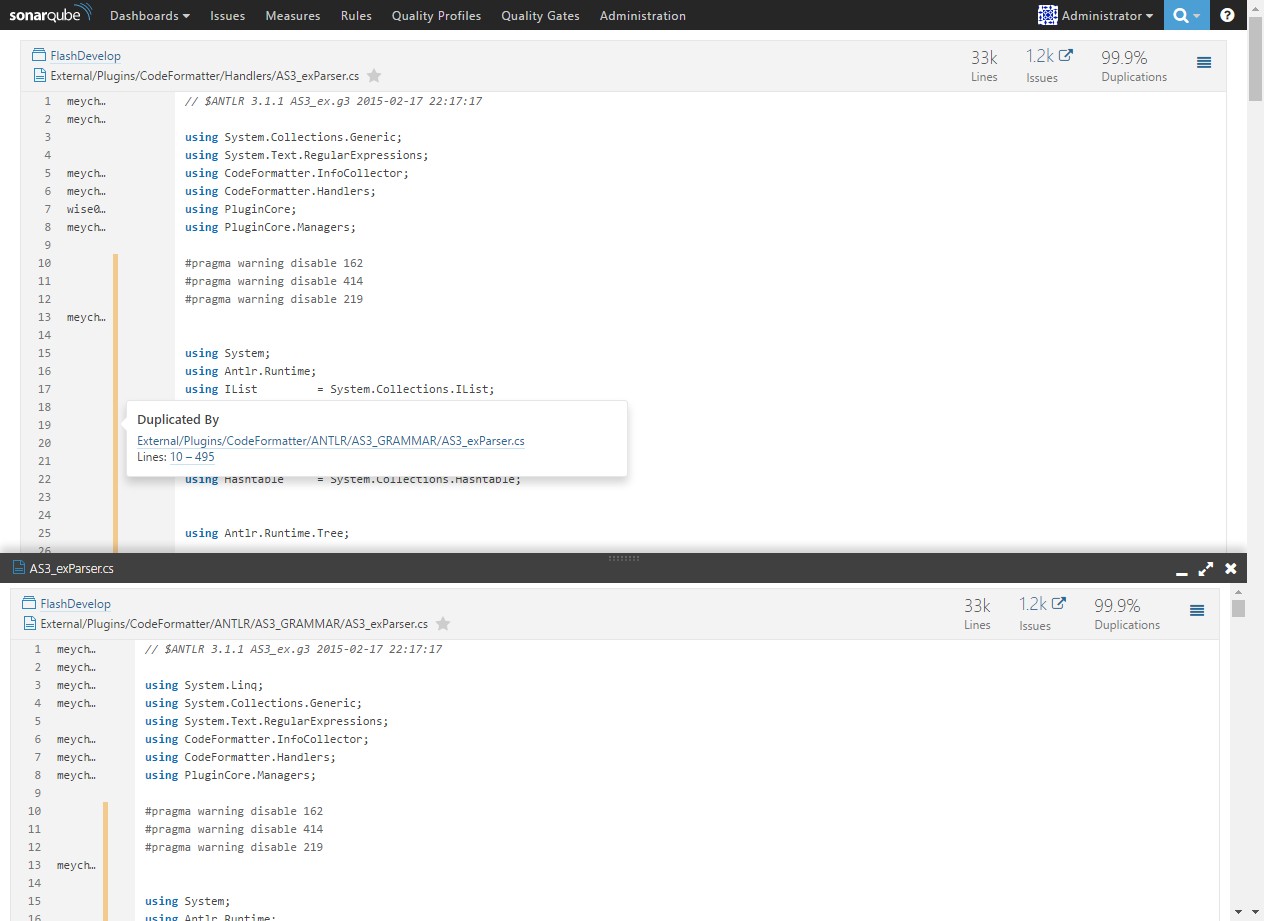
На странице Coverage представлена информация о покрытии кода тестами:



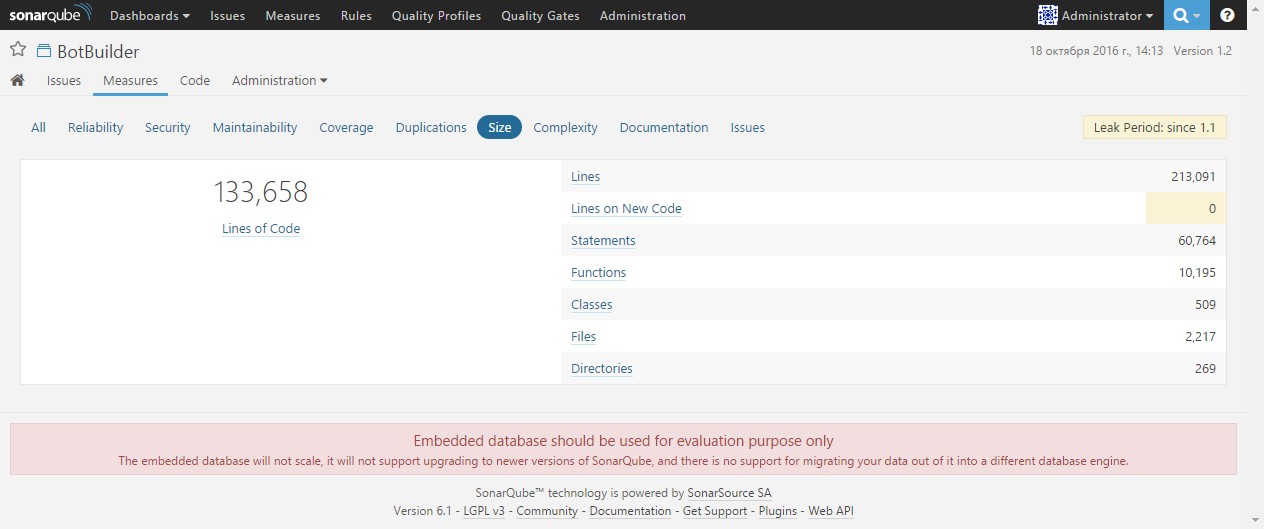
Страница Duplications содержит информацию о дублировании кода в проекте:



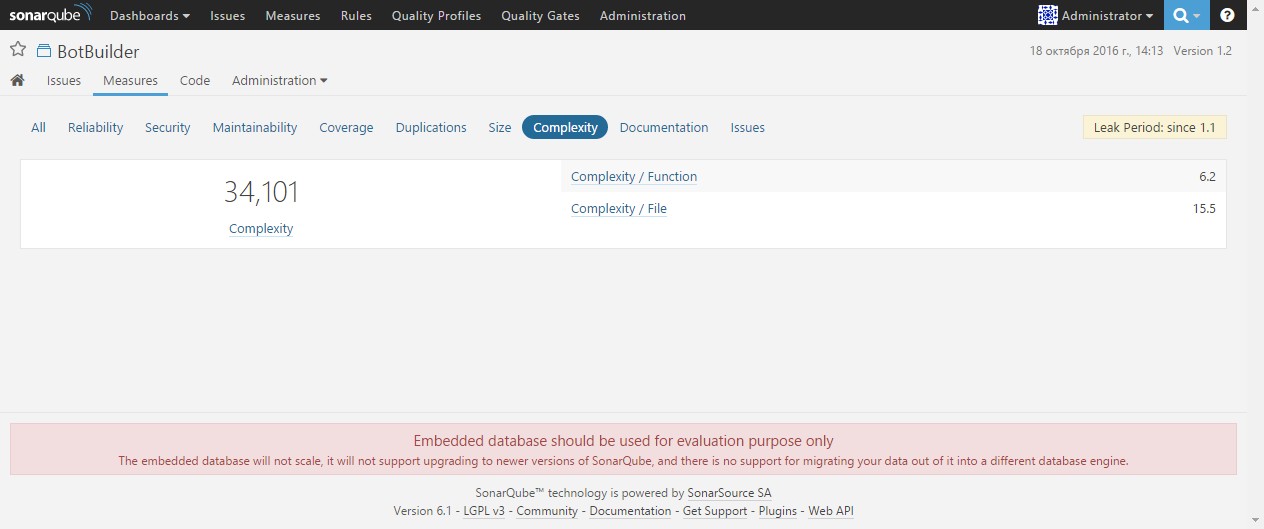
С помощью этой метрики вы легко можете обнаружить повторяющиеся строки, блоки кода и даже целые файлы:



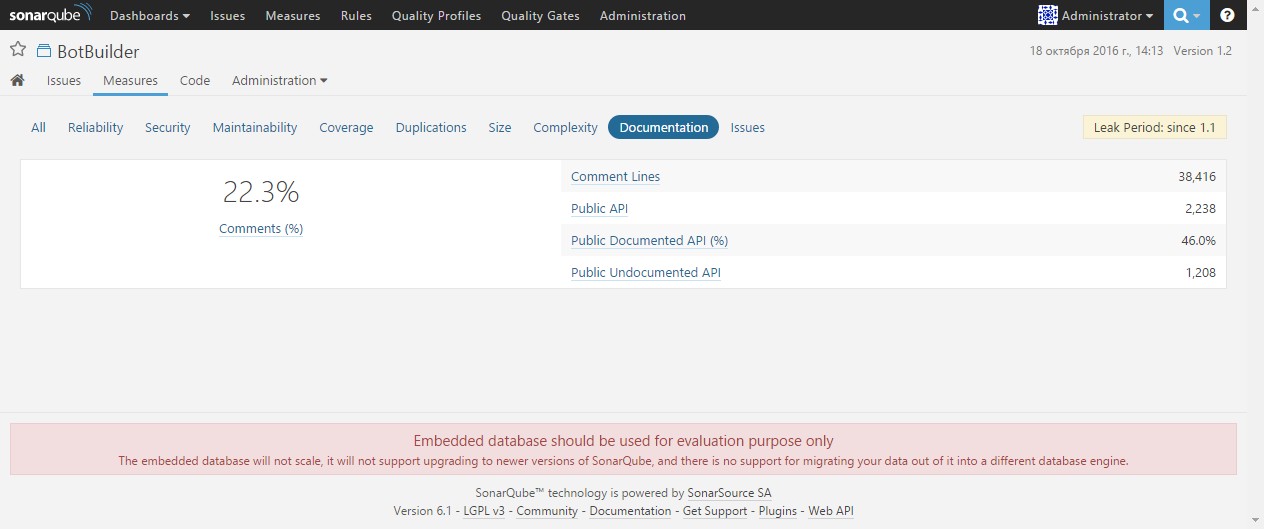
Страница Size содержит информацию о размере проекта: количество строк кода, выражений, функций, классов, файлов и директорий:



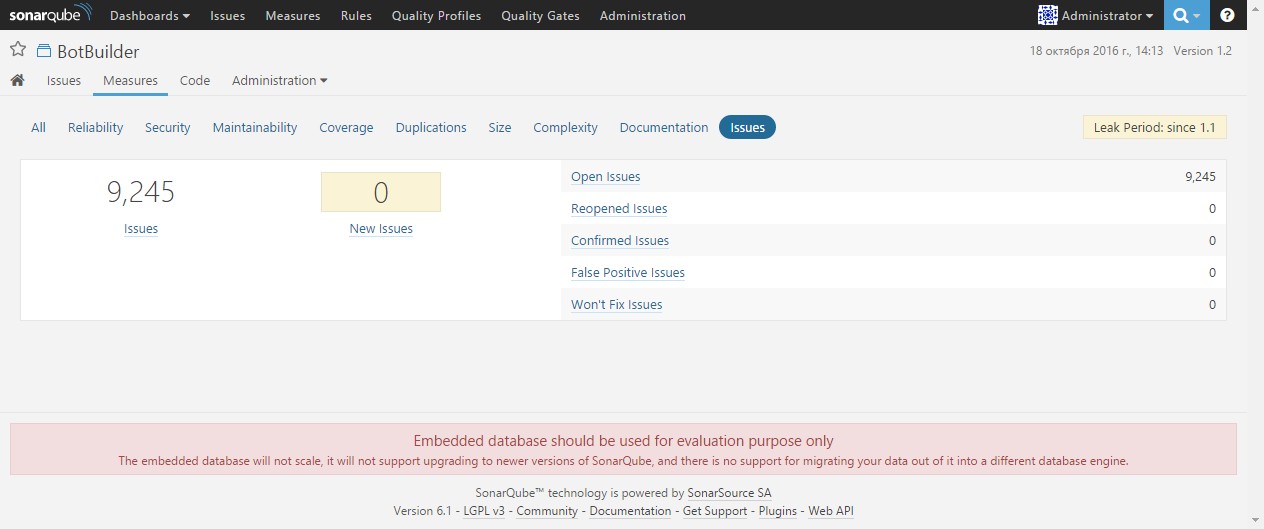
На странице Complexity представлена информация о суммарной цикломатической сложности проекта, а также о средней сложности функций и файлов:



Страница Documentation предоставляет информацию о комментариях в коде: отношение строк с комментариями к общему количеству строк в проекте, количество строк с комментариями, количество публичных API и уровень документирования публичных API:

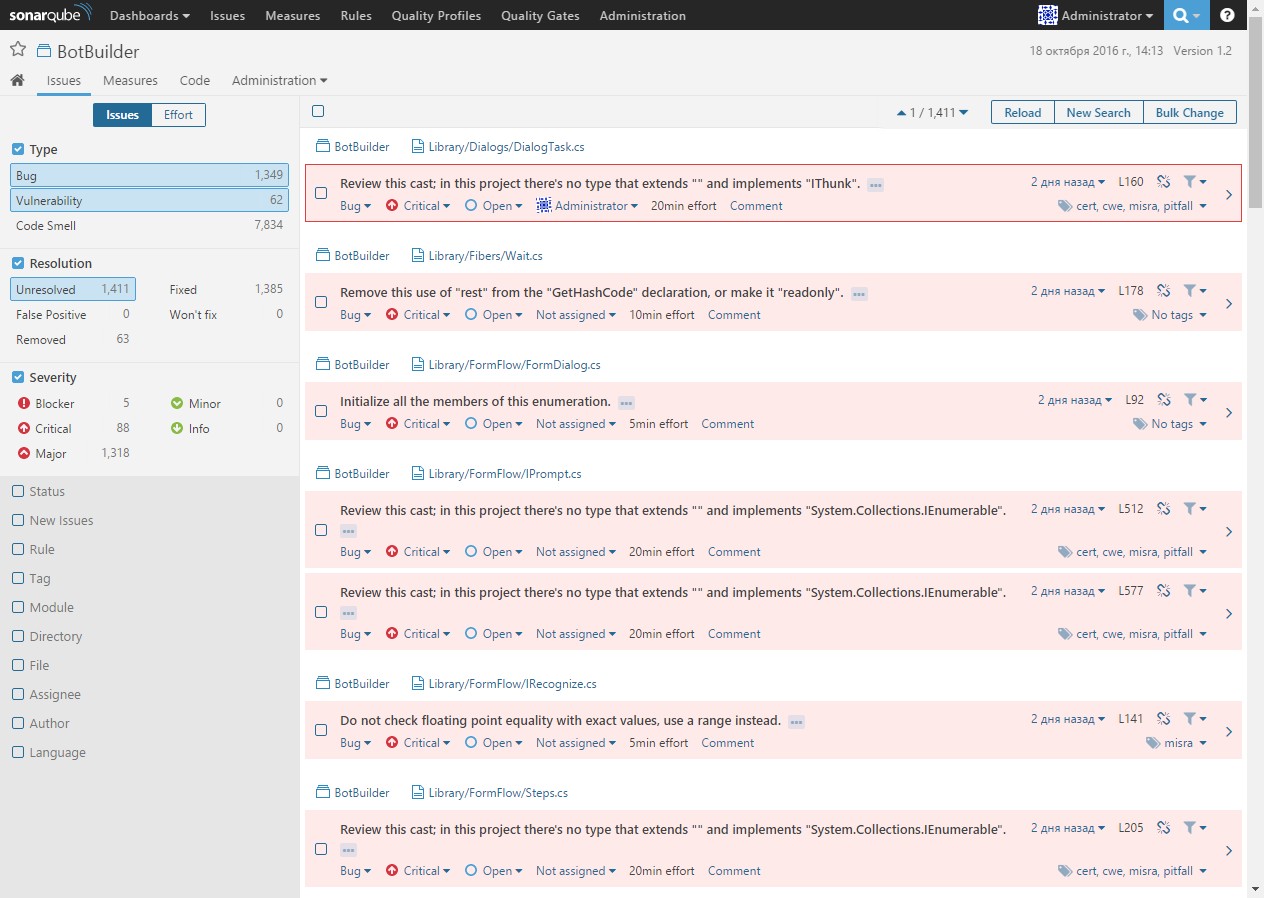


Последняя вкладка в разделе метрик проекта – Issues – содержит общее количество найденных проблем в коде (сумма количества багов, уязвимостей и code smells), а также распределение проблем по состоянию: открытые, переоткрытые, подтвержденные, ложные срабатывания и won't fix:



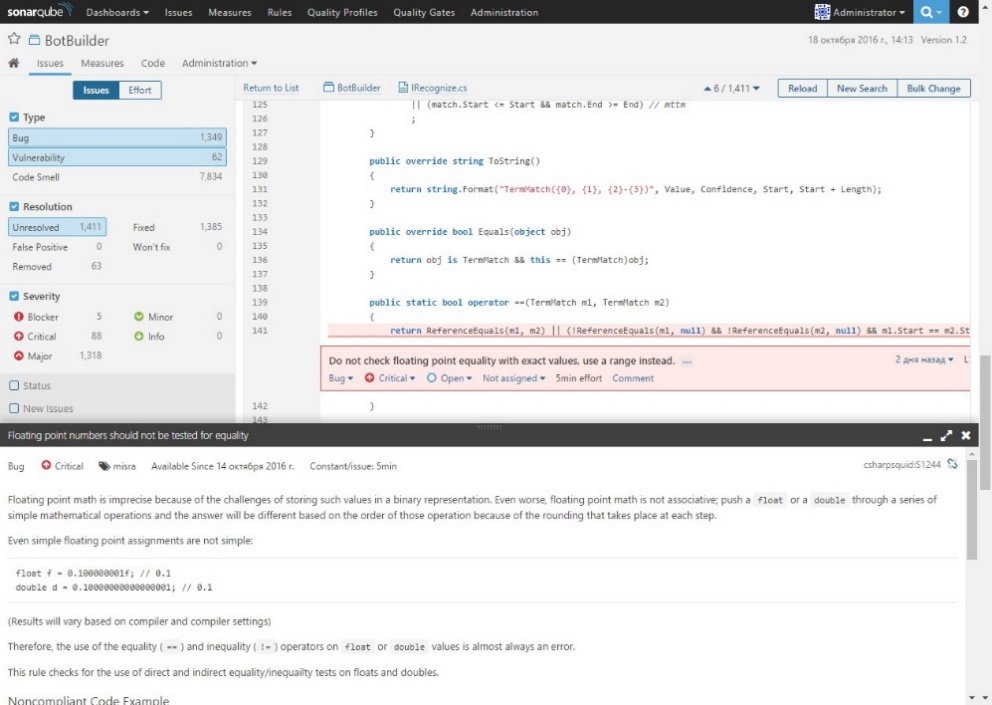
* Навигация по ошибкам и коду

После анализа метрик кода посмотрим, как SonarQube позволяет работать с найденными проблемами в коде. Для этого перейдем в раздел Issues:

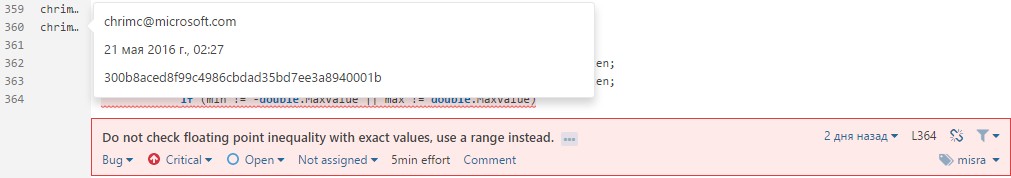


Здесь представлены все найденные проблемы в коде с широкими возможностями фильтрации, что позволяет сфокусироваться на наиболее важных проблемах. Следует отметить, что SonarQube позволяет сохранять настройки фильтров, чтобы повторно их использовать.

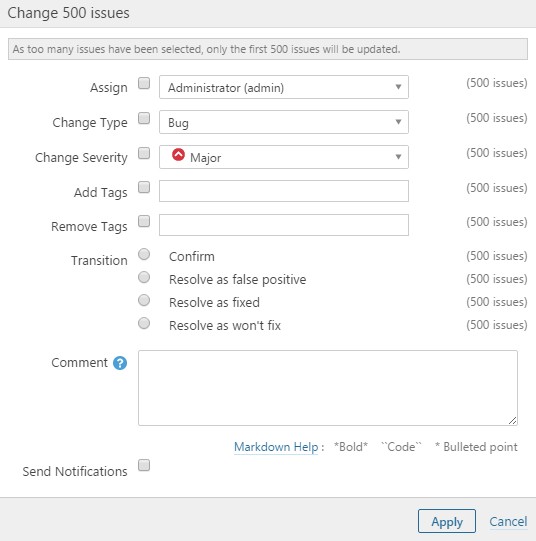
По двойному клику на сообщении об ошибке вы можете перейти к коду, в котором была найдена проблема. Также доступно детальное описание ошибки и рекомендации, как ее исправить:



Обратите также внимание, что, благодаря интеграции с системами контроля версий, видно, кто и когда внес изменения в код, вызвавшие срабатывание анализатора:



Интеграция с системами контроля версий позволяет также автоматически назначать баги в SonarQube на тех разработчиков, которые их допустили. Также вы можете назначать баги на разработчиков вручную, изменять их тип (bug, vulnerability или code smell), важность, теги, добавлять комментарии. Для большего удобства использования доступна функция массового изменения багов:



* Rules, Quality Profiles и Quality Gates

Диагностические правила (Rules), профили качества (Quality Profiles) и границы качества (Quality Gates) – ключевые понятния платформы SonarQube. Каждый плагин для SonarQube, осуществляющий статический анализ кода, содержит репозиторий с описанием диагностических правил, которые этот плагин выполняет. Нарушения этих правил используются для определения технического долга в коде и вычисления времени на устранение проблем. Для удобства использования правила объединяются в профили качества (Quality Profiles).

По умолчанию, SonarQube создает дефолтный профиль качества для каждого поддерживаемого языка, но вы можете создавать свои профили качества с тем набором диагностических правил, которые вам могут быть полезны. Например, для анализа критически важных проектов, требования к качеству кода которых самые строгие, можно определить профиль качества, содержащий все доступные диагностики, а для менее критичных проектов можно определить менее строгий профиль качества, содержащий только серьезные ошибки, что позволит не отвлекаться на незначительные code smells.

Quality Gate – это индикатор соответствия (или несоответствия) кода проекта заданным пороговым значениям метрик. По умолчанию, все проекты, добавленные в SonarQube, используют стандартный quality gate, в котором определены следующие метрики и их пороговые значения:

* Новые баги = 0
* Новые уязвимости = 0
* Коэффициент технического долга на новом коде <= 5%
* Покрытие нового кода >= 80%

**Контрольные вопросы**

1. Какие действия необходимо выполнить, чтобы определить метрики качества кода?
2. Каким образом определяется покрытие кода?
3. Что такое Black Box?
4. Что такое White Box?
5. Приведите порядок отбора тестов для различных случаев.
6. Перечислите основные атаки на код.
7. Что такое внедрение кода?
8. Какие основные причины появления уязвимостей в коде?

1. Для определения метрик качества кода необходимо выполнить следующие действия:

- Оценить читаемость кода.

- Измерить покрытие кода тестами.

- Анализировать сложность кода.

- Оценить количество дефектов и их тяжесть.

- Изучить соблюдение стандартов кодирования.

2. Покрытие кода определяется с помощью инструментов для анализа кода, которые показывают, какая часть кода была протестирована с помощью модульных тестов. Покрытие кода обычно измеряется в процентах.

3. Black Box (Черный ящик) - метод тестирования, при котором тестировщик не имеет доступа к внутренней реализации программы и проверяет ее функциональность по внешнему поведению.

4. White Box (Белый ящик) - метод тестирования, при котором тестировщик имеет доступ к внутренней структуре программы и может проверять ее на основе исходного кода.

5. Порядок отбора тестов для различных случаев:

- Определение граничных значений.

- Выбор тестов для позитивных сценариев.

- Выбор тестов для негативных сценариев.

- Учет комбинаций условий.

- Приоритизация тестов по важности.

6. Основные атаки на код:

- SQL-инъекции.

- XSS (межсайтовый скриптинг).

- CSRF (межсайтовая подделка запроса).

- Переполнение буфера.

- Утечка информации.

7. Внедрение кода - это процесс внесения изменений в исходный код программы или веб-страницу с целью добавления новой функциональности или исправления ошибок.

8. Основные причины появления уязвимостей в коде могут быть связаны с недостаточной проверкой ввода данных, некорректным управлением памятью, недостаточными правами доступа, а также несанкционированным внедрением кода.