План измерения атрибутов качества программного кода

История изменений

Дата	Версия	Описание	Автор
04.06.2020	0.1.0	Черновик	Karinski, Andrey

Введение

Обзор всего документа, включающий цель, область применения, список определений, ссылки и обзор.

Цель

Цель данного документа.

Для заказчика чрезвычайно важно поддерживать хорошее качество кода его программных продуктов, причем этих продуктов весьма много, у них разные разработчики, разные предметные области и технологии. Все это требует максимальной автоматизации процесса контроля качества кода, так как средствами *code review* такой объем кода проконтролировать не представляется возможным.

В данном документе рассматриваются метрики качества программного кода, применяемые для автоматического контроля кода.

Область действия

Краткое описание области действия Плана Измерений: с какими другими документами он связан, на кого он оказывает влияние и, наоборот, что оказывает влияние на него.

Определения и сокращения

Перечень всех терминов, необходимых для правильной интерпретации содержимого документа. Можно указать ссылку на Глоссарий.

Ссылки

Перечень ссылок на документы, которые упоминаются в других частях Плана Измерений.

- Атрибуты качества

Обзор

Описание содержимого остальной части документа и принципов его организации.

Цели управления

Перечень Целей Программы Измерений относительно проекта с точки зрения атрибутов качества.

Основная цель данного "Плана измерений" - выработать четкие и однозначные критерии качества кода и методику его измерения.

Что же такое качество кода и как его улучшить? Это попытка дать интегральную оценку исходному коду на основе определенного набора метрик и показателей. Поскольку почти каждый разработчик имеет свое собственное определение того, что является хорошим кодом, единого определения качества не существует. Несмотря на это, большинство разработчиков сходятся во мнении о нескольких факторах, которые делают код "хорошим" или "качественным".

Есть несколько способов достичь высокого значения качества кода, например:

- **Регулярный обзор кода (code review)**.
  Принято считать, что профессионалы в области программного обеспечения считают *code review* способом улучшить программный код. О бзоры позволяют разработчикам решать проблемы совместными усилиями и делиться знаниями друг с другом, что улучшает их работу. Кроме того, обзоры обеспечивают соответствие кода принятым стандартам.
- **Функциональное тестирование / TDD (test-driven development).**
  Функциональное тестирование важно, потому что оно поощряет разработчиков с самого начала сосредоточиться на функциональности программного обеспечения, сокращая посторонний код. Целью разработки программного обеспечения является написание приложения, которое обеспечивает именно то, что нужно пользователям. Помимо прочего, TDD позволяет сделать дизайн системы более "естественным" и во многом минималистичным.
- **Очистка / дистилляция требований**.
  Большинство проектов разработки программного обеспечения начинаются с документов, содержащих требования в той или иной форме. Проект с четкими, выполнимыми и строго измеримыми требованиями имеет гораздо больше шансов достичь высокого качества, по сравнению с теми проектами, где требования неоднозначные, плохо определенные и не имеющие строгих критериев завершенности (*def inition of done*).

Тем не менее, вышеперечисленные способы требуют большого внимания к коду, высокой квалификации разработчиков, идеально отлаженных процессов. И даже в этом случае не гарантируют проект от досадных ошибок проектирования и кодирования, нарушения принципов гибкого дизайна, поддерживаемости и сопровождаемости. Однако о проблемах в коде, настоящих и потенциальных, можно узнать на основе ряда синтетических показателей, о которых речь пойдет ниже.

Показатели качества кода

Показатели качества - это субъективные измерения, которые направлены на то, чтобы относительно объективно определить, насколько код "хороший". Ниже перечислены некоторые из них, причем показатели напрямую пересекаются с атрибутами качества ПО FURPS+.

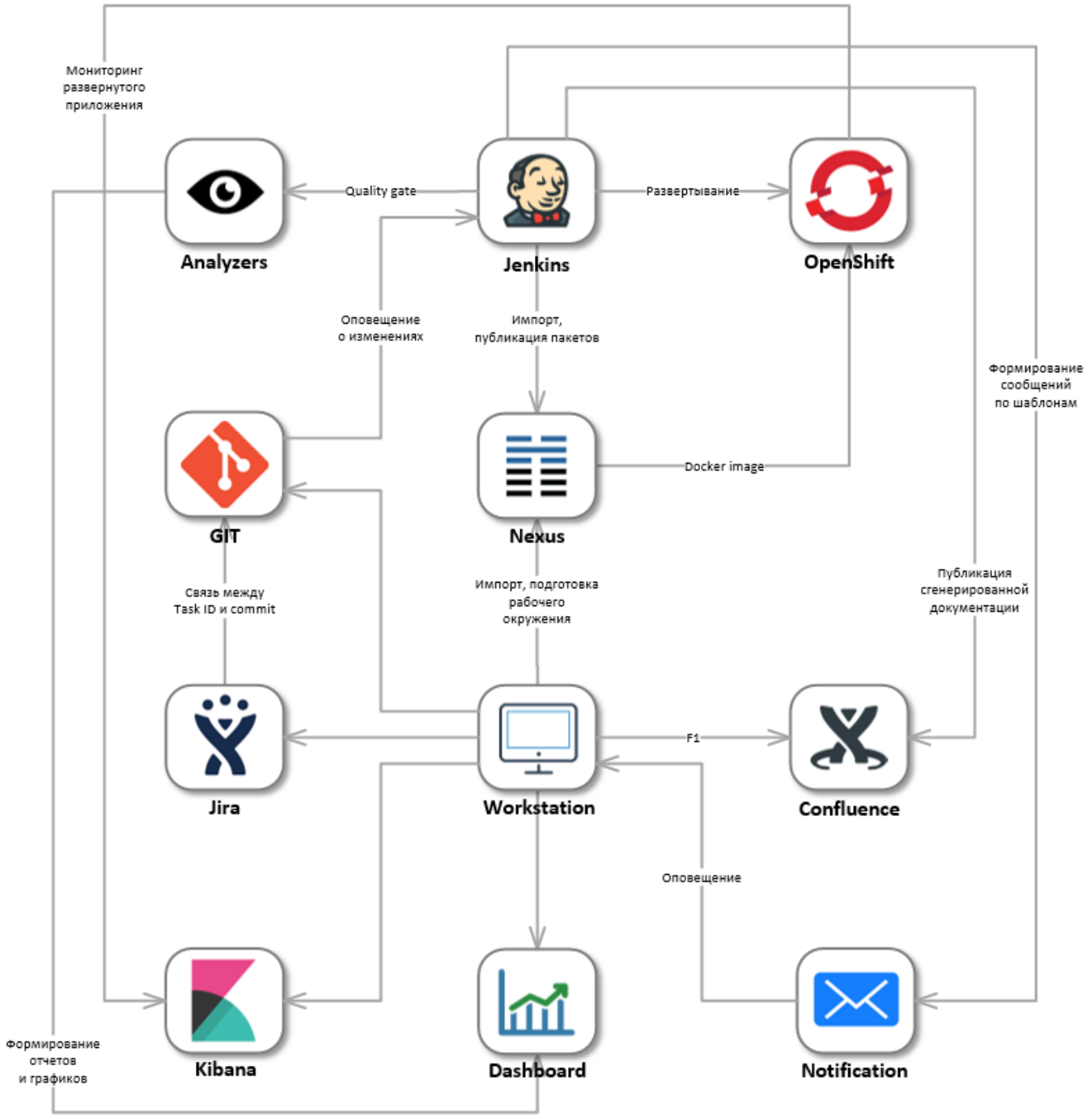
- **Расширяемость**
  Степень или относительная величина, в которой программный код рассчитан на внесение изменений. Основная идея расширяемости исходного кода заключается в том, что разработчики должны иметь возможность добавлять новые функции в код или изменять существующие функции, не затрагивая при этом всю систему.
- **Поддерживаемость**
  Качественное измерение того, насколько легко вносить изменения, и рисков, связанных с такими изменениями. Разработчики могут судить о поддерживаемости следующим образом: если изменение займет час, но в итоге оно занимает три дня, код, вероятно, не так уж и удобен в обслуживании. Еще один способ измерить удобство поддержки - проверить количество строк кода в данной функции программного обеспечения или во всем приложении, так как исходный код с большим количеством строк может быть сложнее поддерживать.
- **Читаемость**
  Читаемый код должен форматироваться в соответствии со стандартами, специфичными для языка, на котором он написан, что делает ст руктуру приложения последовательной и очевидной. Следует помнить, что исходный код читают во много раз чаще, чем пишут, поэтому к читабельности предъявляются особые требования. Комментарии должны использоваться там, где это необходимо, с краткими пояснениями для каждого метода. Имена методов должны быть осмысленными, то есть указывать на то, что делает метод.
- **Ясность, понятность**
  Ясность - это показатель качества, который говорит о том, что хороший код должен быть однозначным. Любой другой разработчик должен иметь возможность легко использовать код, написанный кем-то другим, не тратя много времени, чтобы понять, как он работает.
- **Документируемость**
  Если исходный код не документирован в должной степени, другим разработчикам будет трудно его использовать, или даже самому автору понять код через некоторое время. Документация также предоставляет способ улучшения кода благодаря формализации требований, так как описание причин для определенных программных решений может заметно улучшить дизайн приложения.
- **Тестируемость**
  Хорошо протестированные программы, вероятно, будут более высокого качества, потому что гораздо больше внимания уделяется внутренней работе кода и его влиянию на пользователей. Тестируемость означает, что система постоянно находится под пристальным вниманием и что ее архитектура достаточно гибкая и поддерживает декомпозицию и изоляцию.
- **Эффективность**
  Эффективный код использует только те вычислительные ресурсы, которые ему необходимы. Еще одно измерение эффективности заключается в том, что оно выполняется за минимально возможное время. Некоторое отношение к эффективности также имеет и скорость сборки самого приложения, так как это влияет на эффективность труда разработчиков.

В следующих разделах будет описаны реальные действия, позволяющие более точно и объективно определить соответствие исходного кода перечисленных показателям.

Рабочее окружение

На иллюстрации представлено типичное окружение разработки.

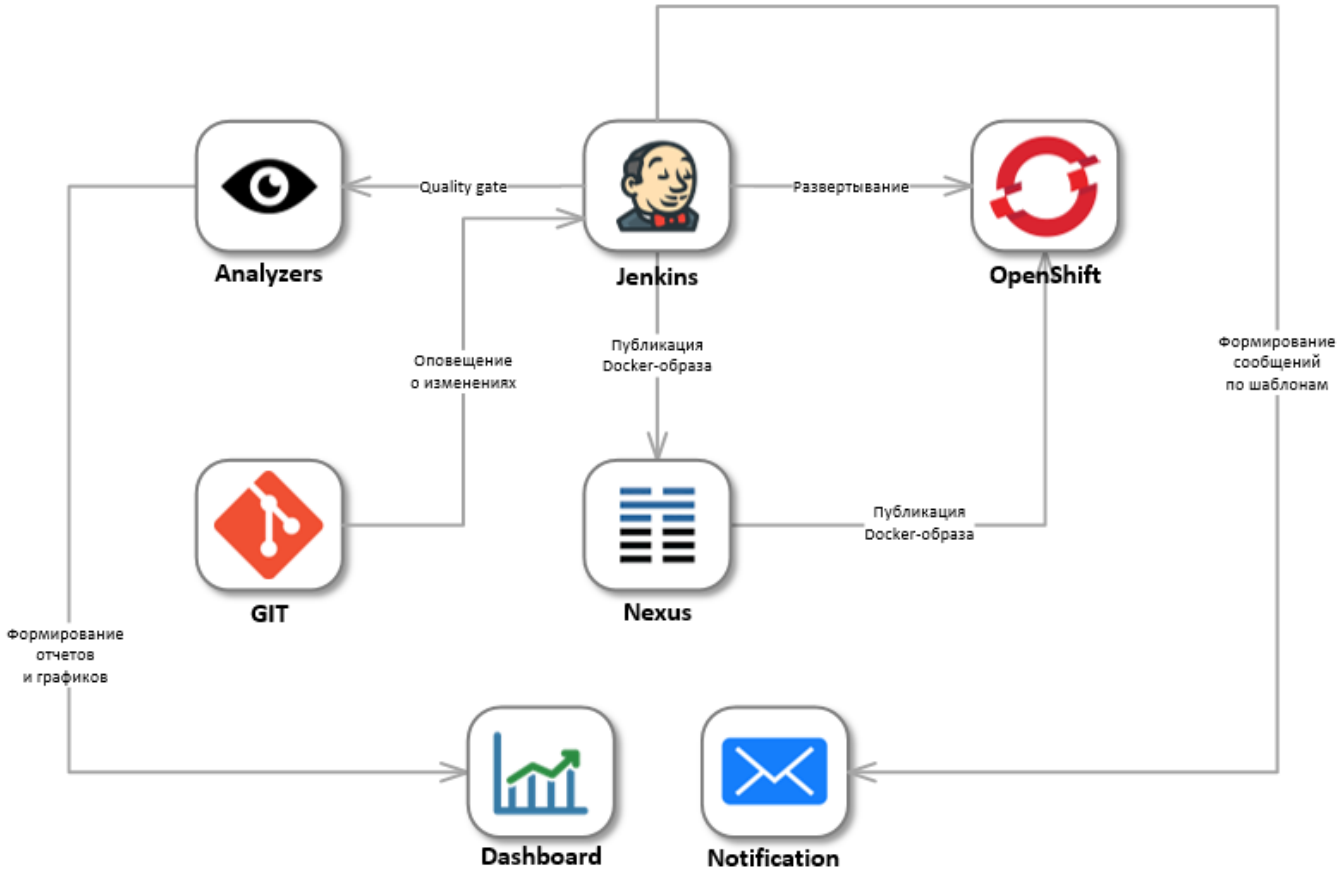
Сервер **Jenkins** забирает исходный код из **git**, передает для анализа в **Analyzers**, где код проходит проверку на предмет соответствия критериям качества (*quality gate*), затем **Jenkins** выполняет сборку, создание образов **Docker**, хранение образов и кодовых пакетов в **Nexus** и развертывание их в **OpenShift**. Кроме того, **Jenkins** обеспечивает автоматическую генерацию документации по коду вместе с публикации ее в **Confluence**, а **Jira** интегрируется с **git** для автоматического трекинга задач. В конце конвейера CI/CD метрики качества попадут в **Dashboard** и отобразятся на ней в различных виджетах. При окончании работы конвейера (успешной или нет) всем заинтересованным лицам рассылаются составленные по специальным шаблонам оповещения (через почту или мессенджеры).



Таким образом, благодаря интеграции отдельных инструментов для решения разных задач цикла разработки, обеспечивается повышение прозрачности самого процесса разработки, понятность текущего состояния проекта, реальные или потенциальные проблемы и их динамика.

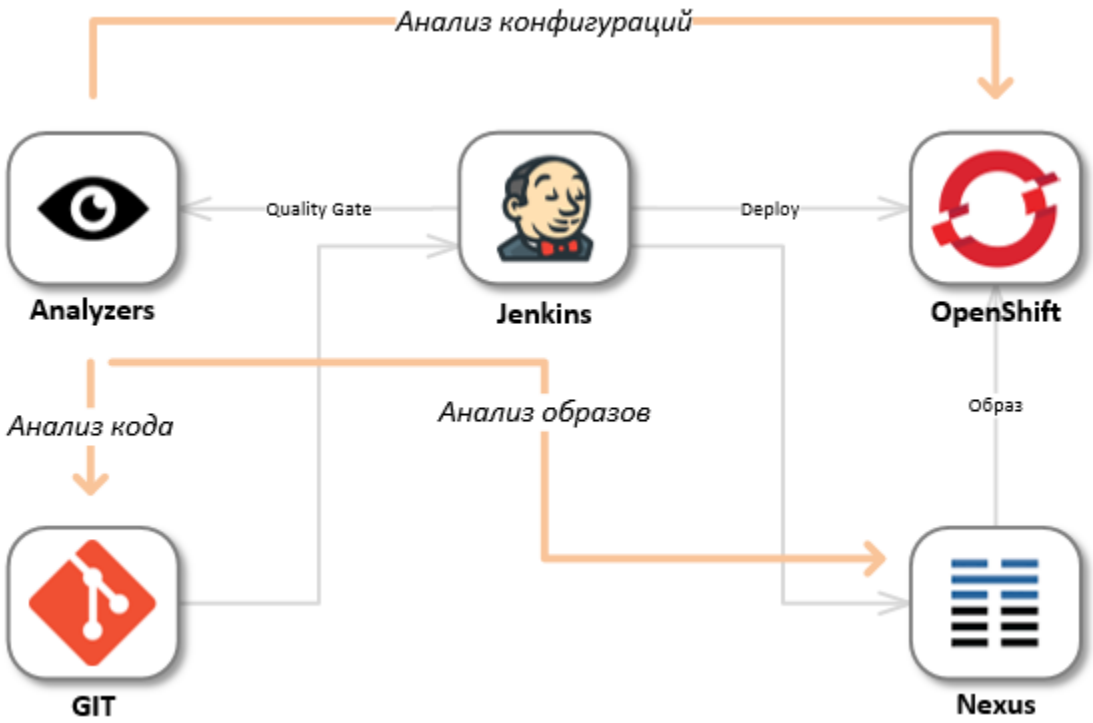
Получение метрик качества

На рисунке представлен фрагмент схемы рабочего окружения, где остались лишь те системы, которые напрямую задействованы в процессе автоматического сбора метрик качества.



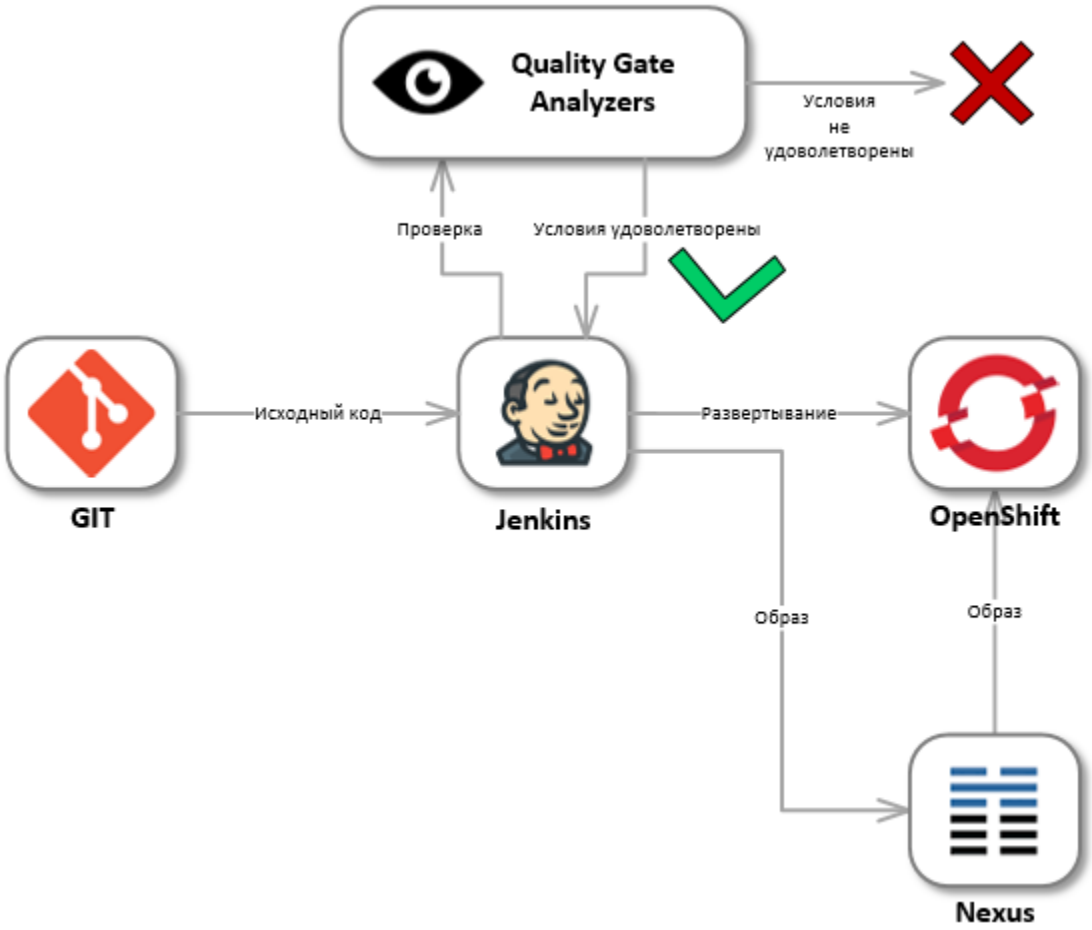
Сборочный сервер **Jenkins** отправляет полученные из разных систем метрики качества и отправляет их для анализа в систему **Analyzers**. Извлекаются следующие типы метрик:

- из **GIT** - метрики качества исходного кода,
- из **Docker** - метрики качества docker-файлов, зависимости и уязвимости образов,
- из **OpenShift** - метрики эксплуатации (скорости отклика компонентов, количество ошибок, время выполнения транзакций),
- из самого **Jenkins** - метрики тестового покрытия, метрики сборки и т.п.



Ворота качества (Quality Gate)

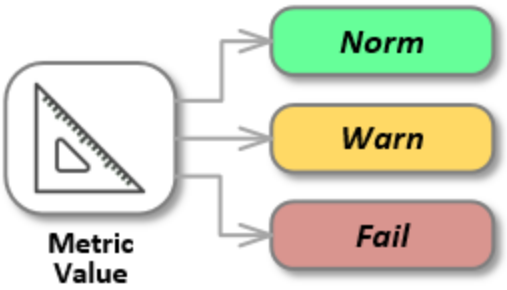
Ворота качества - это важный этап в процессе сборки программного продукта, который требует, чтобы были выполнены заранее определенные критерии, прежде чем проект сможет перейти к следующему этапу сборки. Как правило, *Ворота качества* располагаются после автоматической сборки, но перед ручным code review и последующим за ним успешным окончанием конвейера CI (и, соответственно, началом конвейера CD). *Ворота качества* представляют собой набор автоматизированных проверок попадания значений специально определенных метрик качества в допустимые диапазоны. *Ворота качества* позволяют гарантировать, что проект хорошо продуман с технической точки зрения и может без особых затрат поддерживаться уже после развертывания.



Каждая метрика из общего набора, помимо актуального значения, имеет условные границы ("Норма", "Предупреждение", "Недопустимо") и соответствующие этим границам диапазоны значений:

1. **Pass** - метрика качества в допустимом диапазоне, можно продолжать работу.
2. **Warn** - метрика находится в опасном диапазоне, поэтому связанный с ней код должны быть проверен до продолжения .
3. **Fail** - метрика в недопустимом диапазоне и проблема должна быть решена немедленно.

Зачастую программные проекты не соответствуют времени, бюджету и другим требованиям, но мониторинг качества результатов проекта путем установки контрольных показателей и управления проектом в ключевые моменты может помочь решить эти проблемы.



Например, для метрики "*Коэффициент тестового покрытия*" диапазоны значений могут быть такими:

Граница	Диапазон значений	Реакция Quality Gate
Pass	x > 90%	Прохождение
Warn	50% < x < 90%	Прохождение с соответствующим предупреждением
Fail	x < 50%	Отказ

Итого, каждая метрика для *quality gate* описывается следующими параметрами:

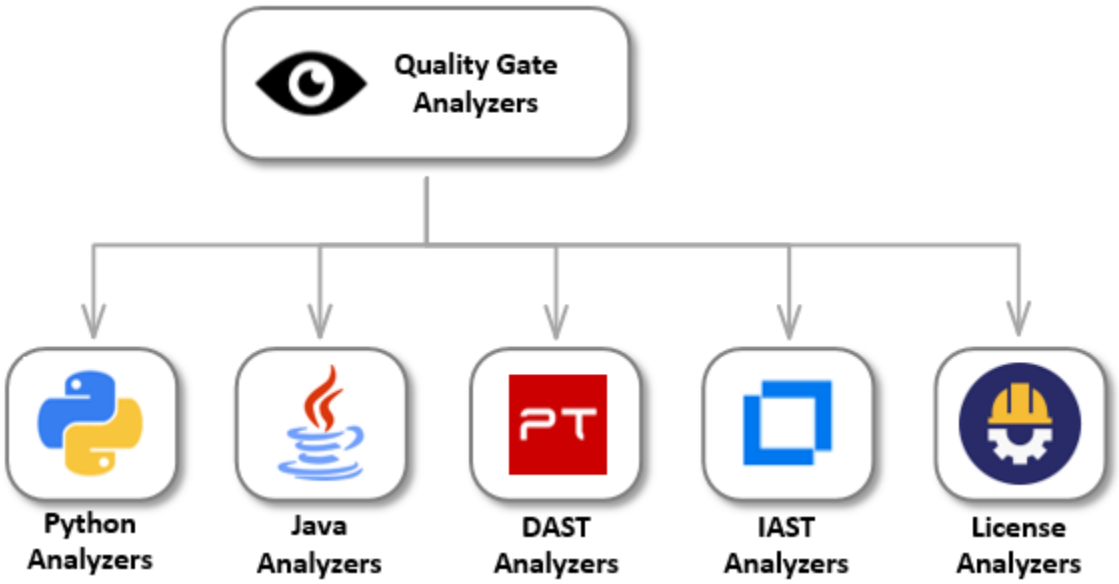
Параметр	Описание	Пример
Measure	Измеренное текущее значение	
Period	Время измерения (относительно предыдущего)	
Operator	Логический оператор или функция для сравнения диапазонов	
Warn Interval	Диапазон предупреждения	
Fail Interval	Диапазон отказа	

Типы и источники метрик

В *воротах качества* анализируются метрики нескольких основных типов:

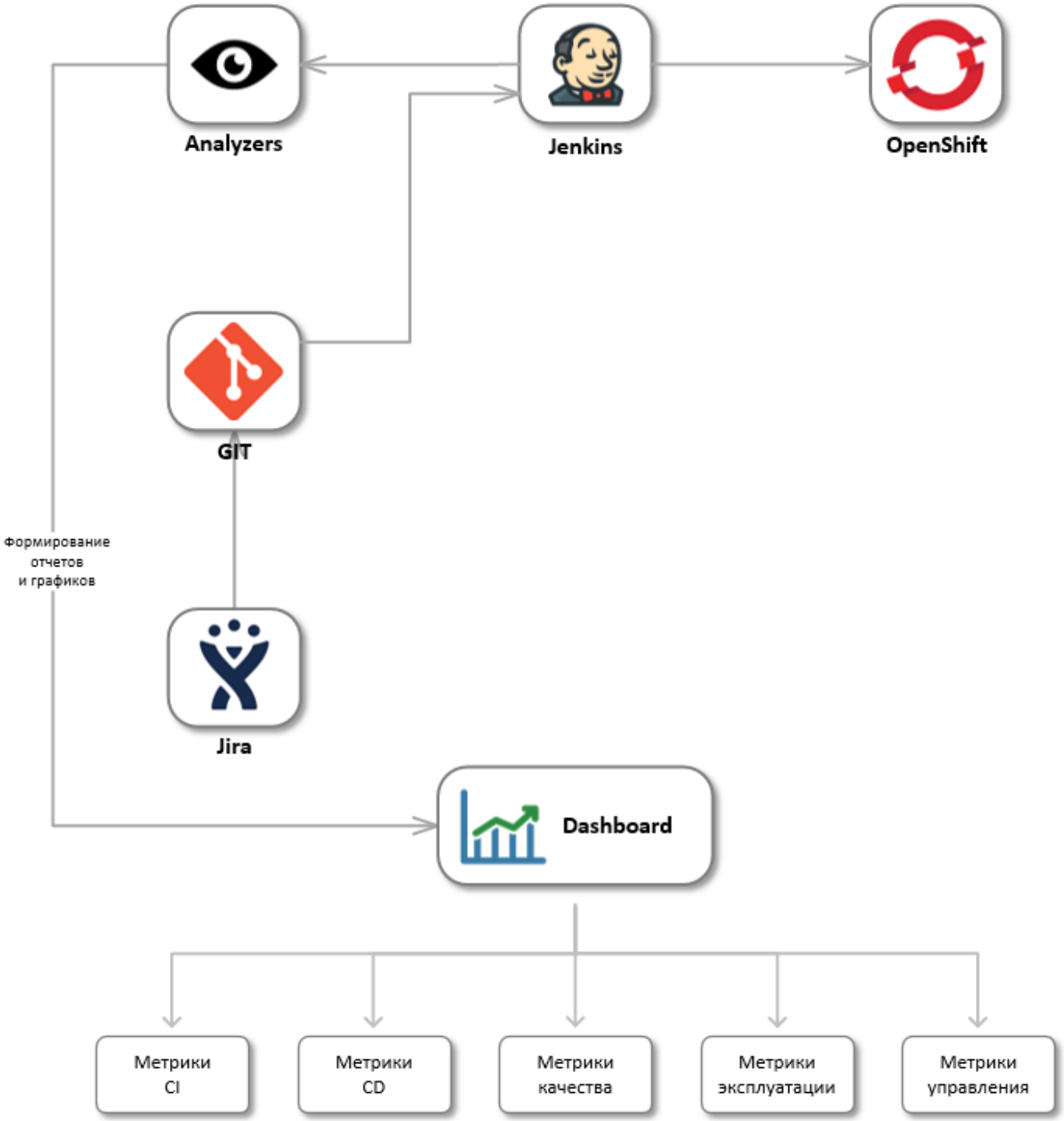
- Метрики качества кода и всего, что относится собственно к коду и платформе.
- Метрики безопасности, то есть уязвимости кода, docker-образов, инфраструктуры, сети и т.д..
- Метрики юридической чистоты - например, соответствие условиям лицензий использованных библиотек.

Следовательно, *ворота качества* смогут автоматически предотвращать многие угрозы безопасности еще до развертывания приложения, что значительно снижает риски и накладные расходы.



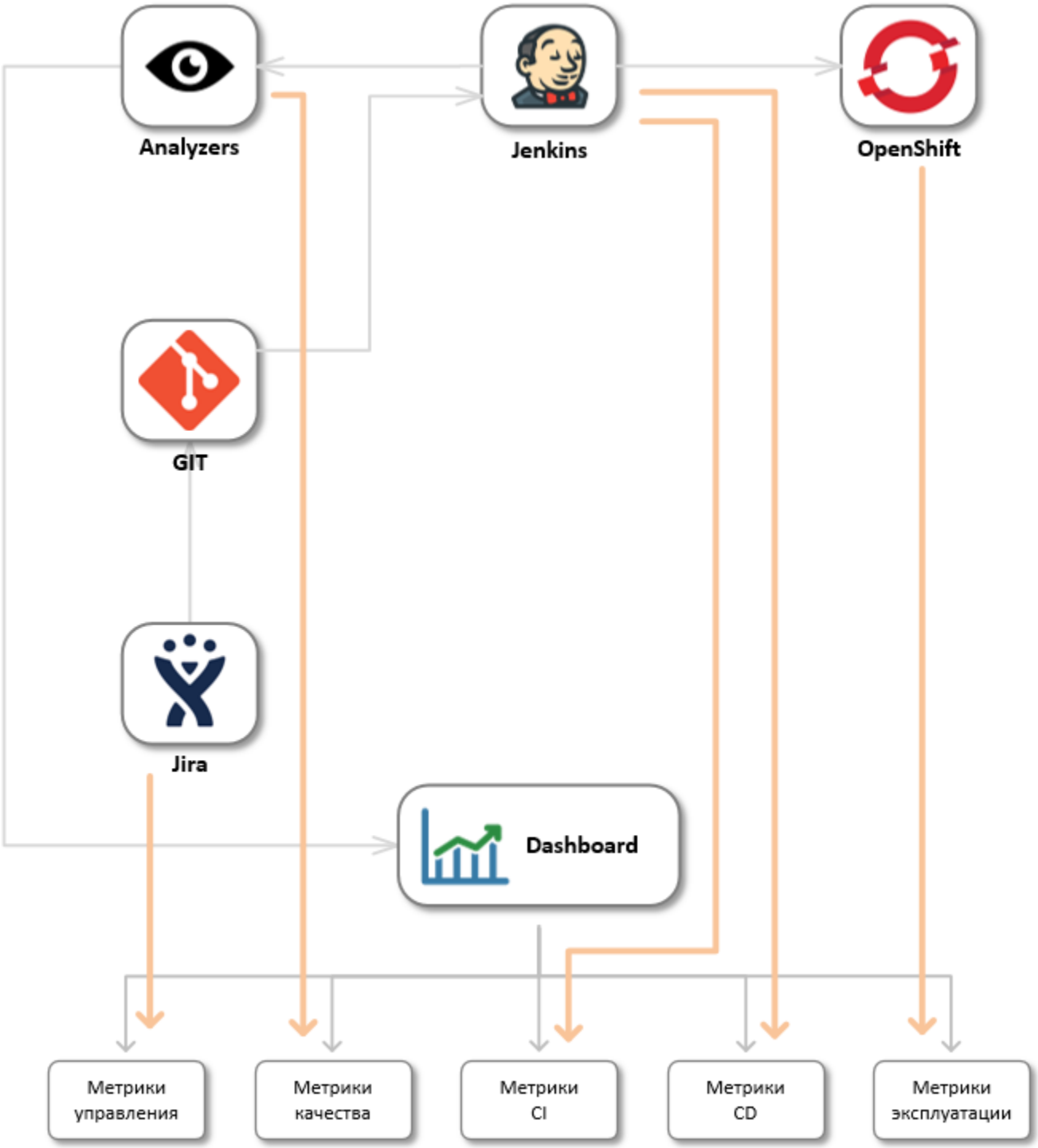
Визуализация метрик качества кода

После сбора и анализа метрики требуется визуализировать в удобной и понятной для всех заинтересованных лиц (членов команды и заказчиков) форме. Для визуализации предполагается создание системы Dashboard.



Как было рассмотрено выше, метрики относятся к нескольким основным группам:

- метрики CI - время сборки, статистика запуска автотестов,
- метрики CD - ошибки, проблемы и показатели развертывания,
- метрики качества - результаты анализа исходного кода,
- метрики управления - статистика по зарегистрированным ошибкам, динамика закрытия задач и их объем,
- метрики эксплуатации - ошибки и измерения из контура промышленной эксплуатации



Существующие решения

На рынке есть несколько готовых решений для реализации *quality gate*, покрывающих своим функционалом большую часть требований:

- JArchitect - для Java
- NDepend - для .NET
- SonarQube - для множества платформ

Специализированные инструменты для конкретных платформ/языков заметно лучше универсальных решений, но универсальные (например, SonarQube) дешевле в использовании.

Каждое из перечисленных решений имеет средства для удобной и гибкой визуализации метрик, что чрезвычайно важно для быстрой реакции на возникающие в проекте проблемы.

Microsoft TFS dashboard

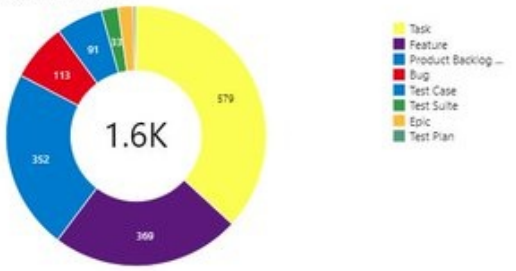
Витрина позволяет выводить данные, касающиеся не только метрик качества, но и бизнес-метрик и метрик управления, а также создавать собственные виджеты.

Work assigned to Srivatsa Marichi (14)

8 Product Backlog Item · 3 Task · 1 Bug · 1 Feature · 1 Epic

ID	State	Title
3217	In Progress	Review application design with technicians.
2516	In Progress	Create service (Add Xamarin Component)
3361	Committed	Calibration UI updates
3365	New	Adding new Task 5/31
3366	New	Supporting platforms are Android and iOS
3367	New	Update the MHC app with latest packages

All Work Items by Work Item Type

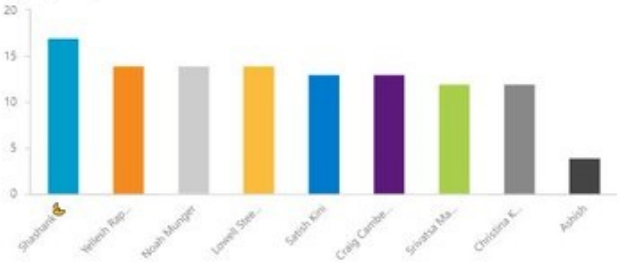


DotNet Team Lead Time

Last 30 days

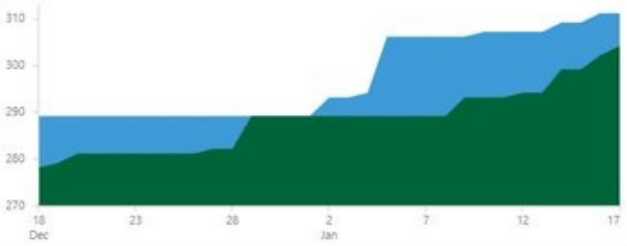
99+ Days on average (Created to Completed) · 36 Product Backlog Item · 19 Bug

Bugs by Assigned To



DotNet Team Backlog items CFD

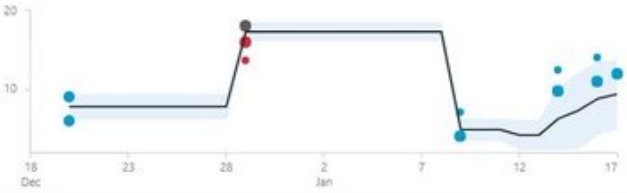
Last 30 days



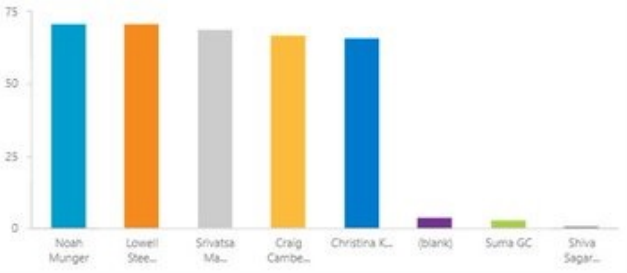
DotNet Team Cycle Time

Last 30 days

13 Days on average (Started to Completed) · 36 Product Backlog Item · 19 Bug



Product Backlog Items by Assigned To



All Work Items (1572)

ID	Work Item...	Title
4351	Task	Update Nuget Packages to the latest version
750	Task	Update patient details
4328	Task	Track number of users for the region" India"
4323	Product Ba...	Running microservices on azure
4327	Task	add microservices for "create exercise"
4322	Product Ba...	AKS integration
4326	Task	AKS integration with Azure
3217	Task	Review application design with technicians.

View query

Pull Requests in MyHealthClinic (3)

View: Assigned to the team

- Merge featureflag to master — Srivatsa Marichi into master, created 3 months ago
- Updated Index.cshtml — Craig Cambell into master, created 3 months ago
- Merge AddingContactUs to master — Christina Kelly into master, created 3 months ago

View summary

Pull Requests in PartsUnlimited (4)

View: Assigned to the team

- Updated StringContainsProductSearch.cs — Srivatsa Marichi into master, created 3 months ago
- Merge DependencyValidation to master — Srivatsa Marichi into master, created 3 months ago
- Completing e2e demo — Srivatsa Marichi into master, created 3 months ago
- Merge AddTerms&Conditions to master — Craig Cambell into master, created 3 months ago

View summary

Team Members

YR

MyHealthClinic

- Caregiver
- Manage your lifestyle

Website Links

- BikeSharing360
- MercuryHealth
- MyHealthClinic

Build Usage

44 minutes this month

BikeSharing360

0 Commits in last 7 days

Bugs

113 Work items

Feature · Epic · Test Plan

Pull Requests in BikeSharing360 (3)

View: Assigned to the team

- Typescript update to version 2.0.5 — Srivatsa Marichi into master, created 3 months ago
- Merge packagesupdate to master — Srivatsa Marichi into master, created 3 months ago
- Merge Ready2017 to master — Srivatsa Marichi into master, created 3 months ago

PartsUnlimited

- A Fabrikam Subsidiary
- Find the parts for you

BikeSharing360

- Book
- Discover
- Enjoy

MercuryHealth

Power BI

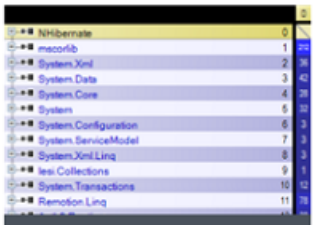
NDepend/JArchitect dashboard

Оба инструмента (фактически разновидности одного решения) умеют выполнять глубокий анализ приложения на предмет всего, что касается только кода, обладают специализированным внутренним языком запросов, позволяющим строить произвольные проекции для метрик.

Diagrams

Dependency Graph

View as SCALED FULL



Dependency Matrix

View as SCALED FULL

Treemap Metric View

View as SCALED FULL

Abstractness vs. Instability

View as SCALED FULL

Application Metrics

Note: Further Application Statistics are available.

Lines of Code

72 627 ▲ +72

15 452 (NotMyCode) ▲ +133

Estimated Dev Effort 1 869d ▲ +1.84d

Types

2 393 ▲ +11

1 Assemblies = no diff

97 Namespaces = no diff

20 559 Methods ▼ -71

5 700 Fields ▼ -108

1 692 Source Files ▼ -2

2 227 Third-Party Elements ▲ +55

Comment

34.71% ▼ from 34.92%

38 608 Lines of Comment ▼ -315

Debt

14.34% ▲ from 12.81%

Rating C 81d effort to reach B

Debt 267d ▲ +28d

Annual Interest 251d ▲ +98d

Breaking Point 12m ▼ -6m

Coverage

75.7% ▲ from 75.49%

54 979 Lines of Code Covered ▲ +207

17 648 Lines of Code Not Covered ▼ -135

0 Lines of Code Uncoverable = no diff

Method Complexity

192 Max = no diff

1.99 Average ▼ -0.048

Quality Gates

Fail	5
Warn	4
Pass	2

Rules

Critical	12	
Violated	101	+1
Ok	41	

Issues

All	16 199	+3 435	-1 733
Blocker	0		
Critical	4	+3	
Major	2 931	+2 058	-103
Minor	11 926	+1 256	-1 466
Info	1 338	+118	-164

Quality Gates summary

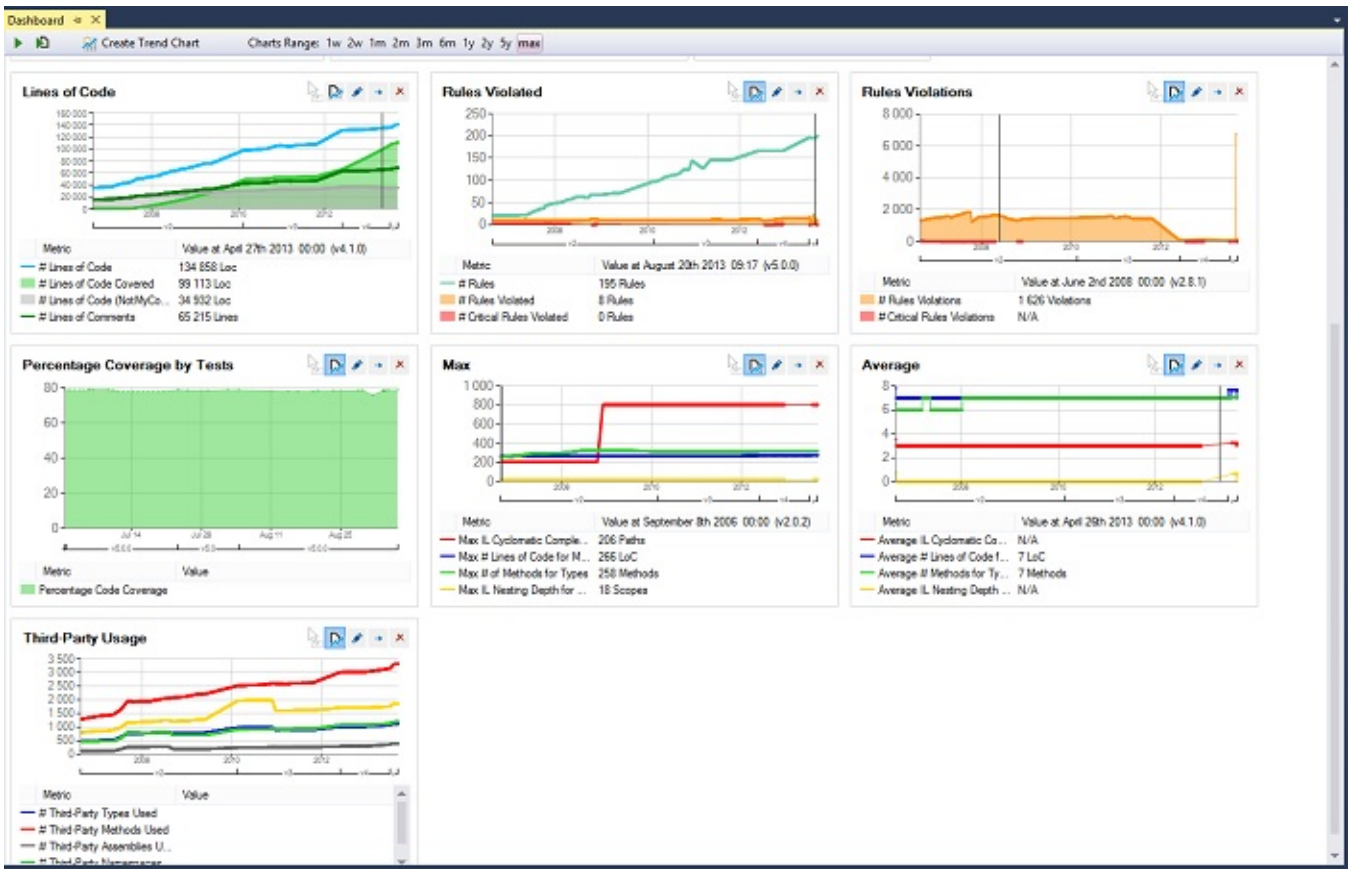
2 4 5

Some Quality Gates fail. The build can be stopped upon quality gate failure. Online documentation.

Quality Gates that measure diff cannot be run on the baseline. Hence they have blank trend and baseline status.

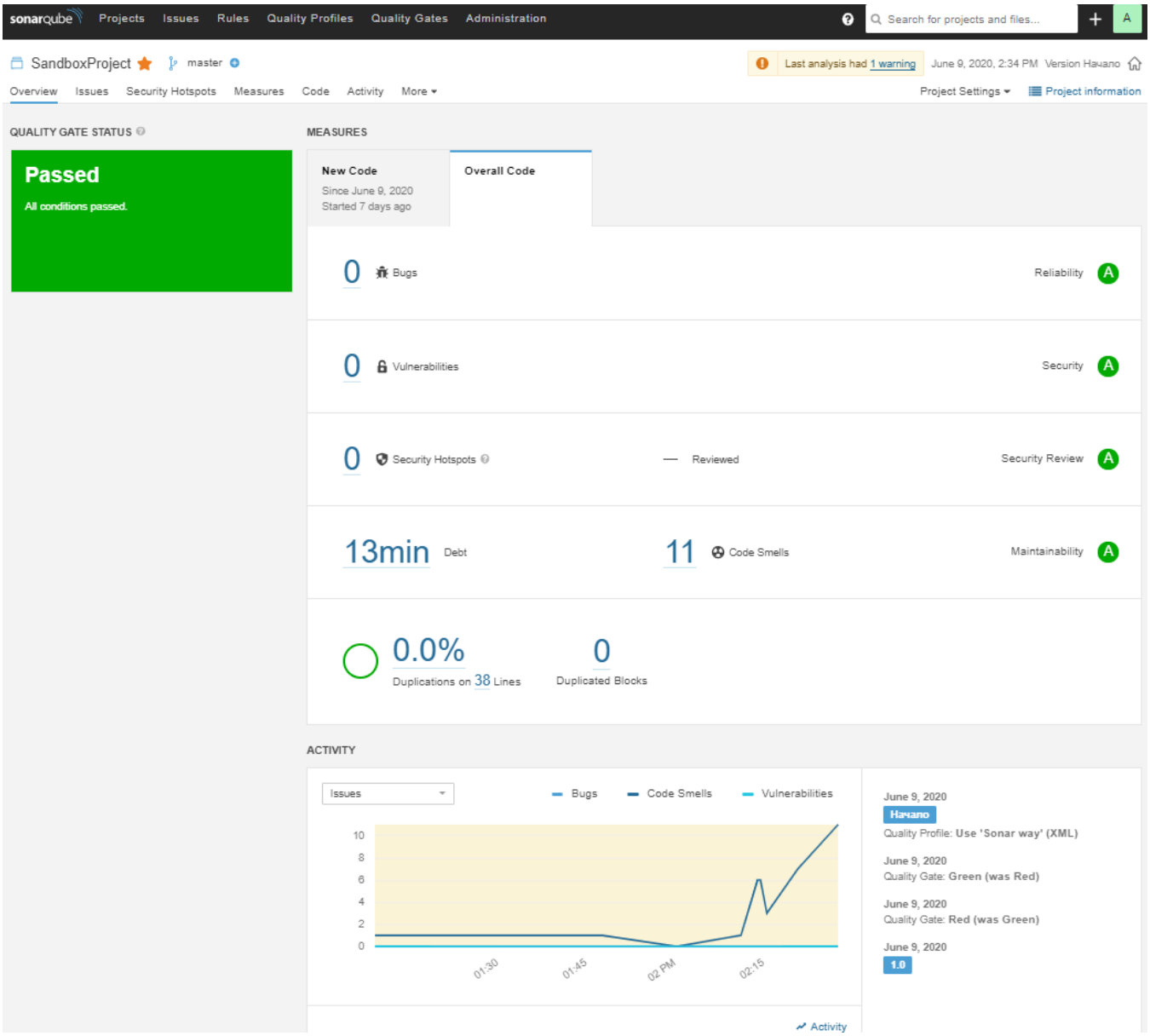
Name	Trend	Baseline Value	Value	Group
Percentage Coverage	▲	75.49 %	75.7 %	Rules in : NDepend_v2017_1_0_DefaultRules.ndrules \ Quality Gates
Percentage Coverage on New Code			72.95 %	Rules in : NDepend_v2017_1_0_DefaultRules.ndrules \ Quality Gates
Percentage Coverage on Refactored Code			76.43 %	Rules in : NDepend_v2017_1_0_DefaultRules.ndrules \ Quality Gates
Blocker Issues	=	0 issues	0 issues	Rules in : NDepend_v2017_1_0_DefaultRules.ndrules \ Quality Gates
Critical Issues	▲	3 issues	4 issues	Rules in : NDepend_v2017_1_0_DefaultRules.ndrules \ Quality Gates

Важно визуализировать не только статичные показатели, но и тренды. Пример ниже показывает, как рисуются графики некоторых проектным метрик.



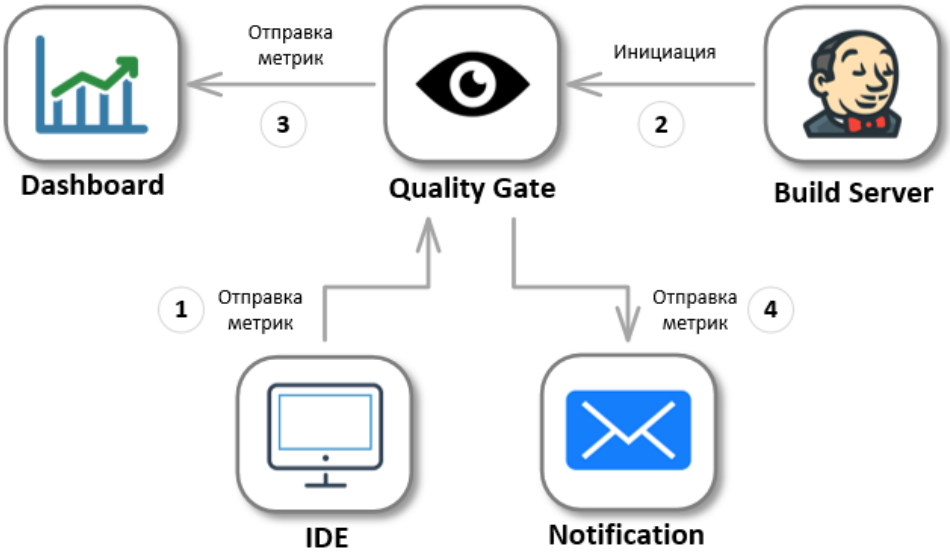
SonarQube dashboard

Витрина sonarqube заметно скромнее по возможностям, но зато сам sonar поддерживает в той или иной степени много платформ и языков, а простота интеграционного API позволяет сделать "мостики" для произвольных, платформенно-специфичных анализаторов.



Специализированное решение

Так как не одно из существующих на рынке решений *quality gate* не удовлетворяет в полной мере потребностей заказчика в области сбора и визуализации разнообразных метрик, имеет смысл разработать специализированное решение. Для экономии ресурсов будет разумным использовать в качестве базы *quality gate* систему **SonarQube**, создав для него дополнительные плагины для произвольных метрик. Из тех же соображений роль службы оповещения (о выходе метрик из заданных диапазонов и других событиях) может быть возложена на компонент **Prometheus AlertManager**, а в роли *витрины* (*dashboard*) можно использовать **Kibana** из состава **ELK**. Все предложенные компоненты входят в целевую экосистему ГПН, что упростит их ввод в эксплуатацию и сопровождение.



Детали предполагаемой реализации

На UML-диаграмме видны пакеты, представляющие основные подсистемы *ворот качества*. Краткое описание их содержимого и обоснования ниже.

Пакет Quality Gate

Пакет содержит основную логику сбора и последующего анализа различных метрик.
Ядро всей подсистемы - **SonarQube**, используемый как сервис посредством предоставляемых им публичных API. Сервис **SonarQube** имеет встроенный набор анализаторов метрик качества кода, а также кастомизируемых правил анализа, которые будут использованы для построения финального отчета. Для анализа метрик, для которых у **SonarQube** не предусмотрено источников, существует **Plugin API**, позволяющий, как ясно из названия, создавать плагины на языке **Java**. Например, можно создать плагин для интеграции с Jira для получения метрик управления (производительности команды разработки, соотношения открытых и закрытых багов, т.д.). Разумным будет создание плагинов-адаптеров к специализированным инструментам на **JS/TS, Python, SQL**.

Пакет IDE

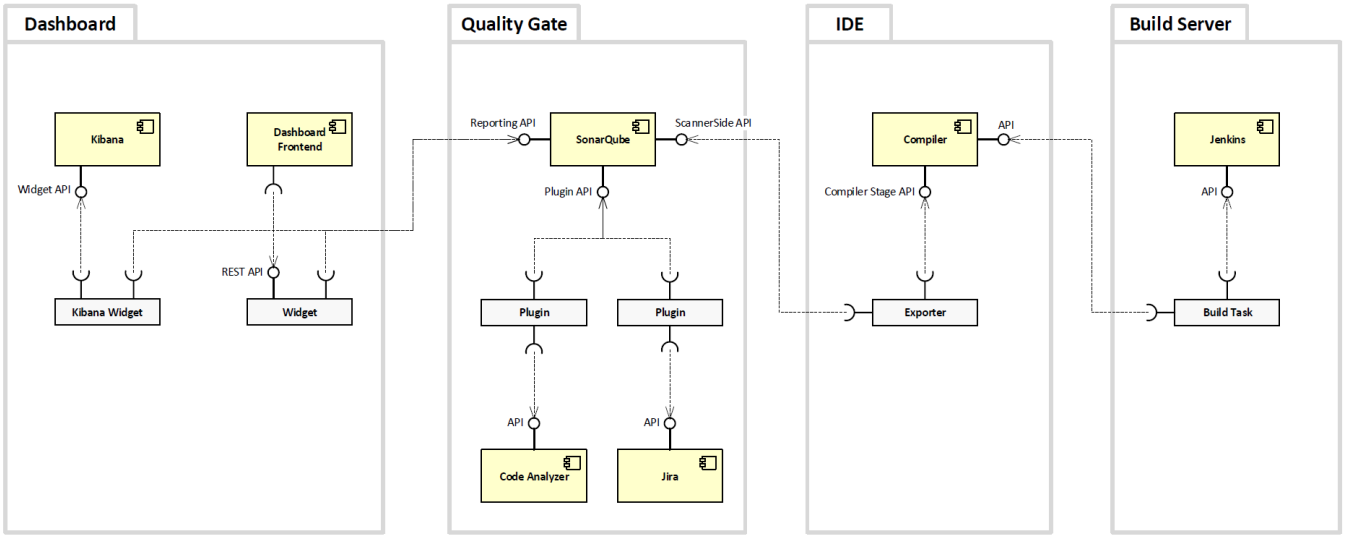
Многие IDE имеют возможность создавать плагины, интегрирующиеся в процесс парсинга/компиляции и умеющие выполнять анализ и сбор метрик качества кода собственными силами. Как правило, такие инструменты наиболее совершенны, так как выполнялись строго специализированными под язык или фреймворк.
Пакет **IDE** содержит специальный адаптер (**Exporter** на диаграмме), обеспечивающий интеграцию таких анализаторов с **SonarQube** через специализированный интерфейс **ScannerSide API**, существующий для подобных задач. Таким образом, **SonarQube** получает обобщенным образом метрики из плагинов IDE.

Пакет Build Server

В процессе сборки **Jenkins** использует сборочные сценарии, предполагающие вызов анализаторов кода из **IDE**.

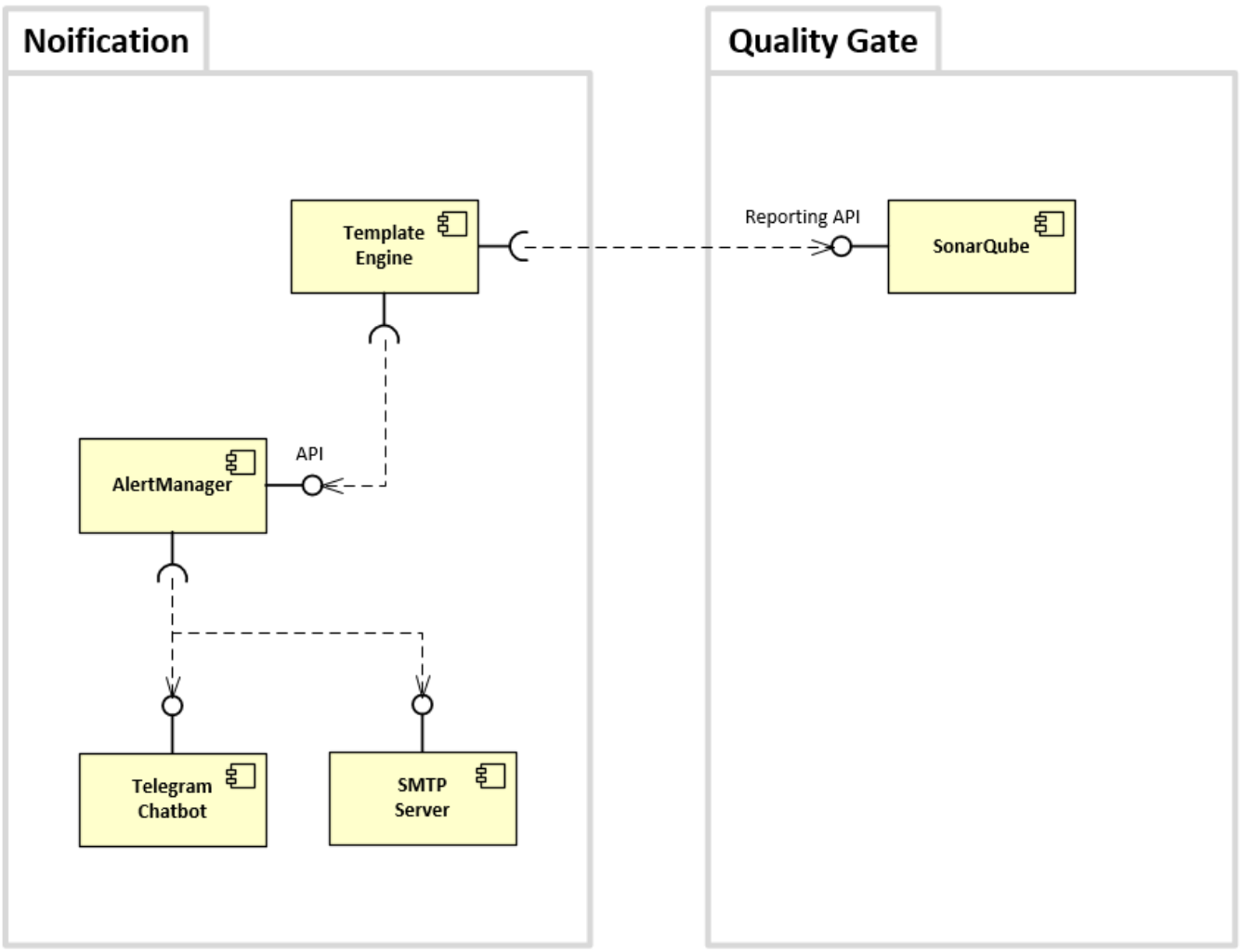
Пакет Dashboard

Пакет содержит все, что нужно для создания гибкой и универсальной витрины данных, отображающей собранные подсистемой **Quality Gate** метрики. В общем случае предполагается создание набора специализированных виджетов, отрисовывающих нужные метрики, получаемые из **SonarQube** через интеграционный интерфейс **Reporting API**. Каждый виджет должен быть кастомизируемым: уметь менять стили отрисовки, масштаб, фильтры и так далее. В зависимости от дальнейших потребностей ГПН, реализация витрины может базироваться на системе **Kibana** либо быть полностью самостоятельным решением.



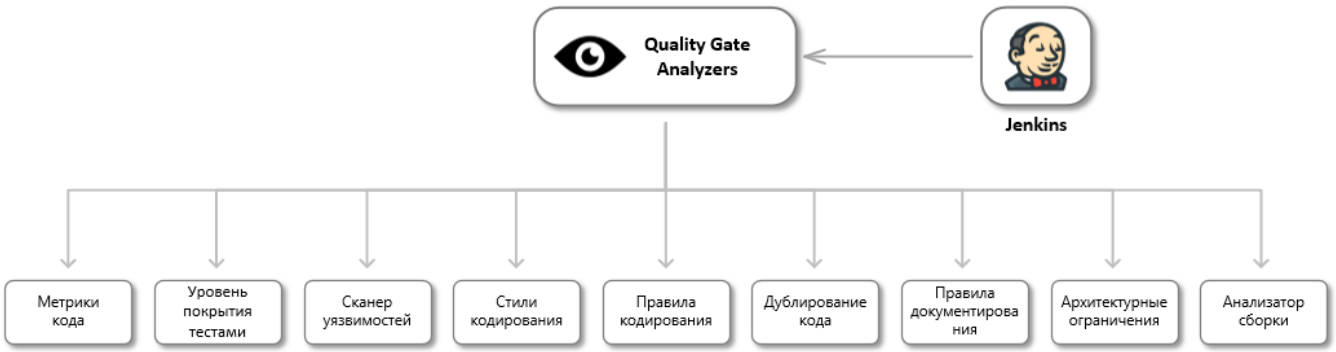
Пакет Notification

Предполагается, что подсистема Notification будет применяться для массовых оповещений всех заинтересованных в процессе сборки лиц. Следовательно, разумно построить такую систему на базе существующего, развернутого и настроенного компонента Prometheus AlertManager, который будет использоваться не напрямую, а посредством специального компонента Template Engine, то есть генератора тела сообщений по заданным шаблонам. Такой генератор, подписавшись на события SonarQube посредством Reporting API, будет формировать и отправлять оповещения по различным каналам в зависимости от различных условий (например, сообщение в Telegram при ошибке сборки, письмо с подробным отчетом и так далее).



Метрики

Перечень метрик, которые должны определяться на основании Целей Программы Измерений и собираться с заданной периодичностью.



Ниже представлен неполный перечень метрик, которые предполагается собирать для обеспечения работы *quality gate*. Рекомендуется проанализировать как сам этот перечень, так и формат описания метрик.

Метрики качества

Метрики кода

За годы развития индустрии было разработано множество показателей для анализа исходного кода программного обеспечения. Они варьируются от самых простых, вроде общего объема исходного кода программы или количества строк кода, до более сложных, таких как цикломатическая сложность и коэффициенты сопровождаемости. Различные метрики имеют смысл на разных уровнях абстракции.

Размерные метрики (Dimensional Metrics)

Этот вид метрик существует для количественного измерения характеристик программного кода с точки зрения его размеров и модульности. Несмотря на то, что большой размер кода может быть признаком множества реализованных функций, фактически это может сигнализировать о более высокой вероятности появления ошибок.

Строки кода (Lines of Code)

Название	**Название метрики и его известные синонимы**
Определение	Общее количество строк кода программного модуля.
Цели	Метрика может быть полезна при сравнении общего размера двух классов, проектов или приложений. Также может быть интересна как условный эквивалент выполненной работы.
Процедура анализа	TODO
Обязанности	TODO

Количество классов в модуле (Number of Classes)

Название	**Название метрики и его известные синонимы**
Определение	Значение количества классов (типов) в программном модуле (пакете).
Цели	
Процедура анализа	TODO
Обязанности	TODO

Количество методов класса (Number of Methods)

Название	**Название метрики и его известные синонимы**
Определение	Количество методов класса.
Цели	Классы со слишком большим количеством методов могут пытаться сделать слишком много, или в любом случае может быть сложнее поддерживать. 20 и более - серьезная причина для расследования.
Процедура анализа	TODO
Обязанности	TODO

Количество полей класса (Number of Fields)

Название	**Название метрики и его известные синонимы**
Определение	Количество полей класса.
Цели	Как и в случае с методами, наличие слишком большого количества полей может указывать на проблему сопровождения. 20 и более - повод для беспокойства.

Процедура анализа	TODO
Обязанности	TODO

Количество параметров (Number of Parameter)

Название	**Название метрики и его известные синонимы**
Определение	Количество параметров метода. Методы со слишком большим количеством параметров сложнее использовать и они часто бывают более сложными.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Количество переменных (Number of Variables)

Название	**Название метрики и его известные синонимы**
Определение	Количество переменных, объявленных в методе, большие числа часто соответствуют методам, которые сложнее поддерживать.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Метрики сложности (Complexity Metrics)

Этот вид метрик оценивает относительную сложность программного кода. Поскольку понимание кода тесно связано с его сложностью, эти показатели помогают понять, насколько дорогостоящим является внесение изменений в код или когда необходимо выполнить полноценное модульное тестирование. Более сложный код имеет больше возможных путей выполнения, поэтому его сложно полностью протестировать, а так же сложнее понять и поддерживать.

Связность классов (Class Coupling)

Название	**Название метрики и его известные синонимы**
Определение	Связность кода.
Цели	Чем больше уникальных типов ссылается на класс, тем менее он стабилен, поскольку любые изменения любого из связанных типов могут нарушить рассматриваемый класс.
Процедура анализа	
Обязанности	TODO

Суммарная сложность типа (Aggregate Type Complexity)

Название	**Название метрики и его известные синонимы**
Определение	Определяет суммарную сложность типа, собирая метрики сложности содержащихся в нем методов.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Суммарная сложность модуля (Aggregate Project Complexity)

Название	**Название метрики и его известные синонимы**
Определение	Определяет суммарную сложность модуля, собирая метрики сложности содержащихся в нем типов.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Объектно-ориентированные метрики (Object-Oriented Metrics)

Поскольку большинство современных приложений реализованы с использованием объектно-ориентированных языков программирования (Java, C#, C++), то стоит оценить качество кода с помощью набора метрик ООП.

Количество потомков (Number of Children)

Название	**Название метрики и его известные синонимы**
Определение	Показывает количество непосредственных подклассов, подчиненных классу в иерархии наследования. Чем больше число потомков, тем больше повторное использование, но может потребоваться дополнительное тестирование методов из-за возможного нарушения LSP.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Глубина наследования (Depth of Inheritance)

Название	**Название метрики и его известные синонимы**
Определение	Глубина наследования.
Цели	Большая глубина наследования указывает на более сложную иерархию объектов, что потенциально может увеличить "хрупкость" кода.
Процедура анализа	TODO
Обязанности	TODO

Недостаток зацепления методов (Lack of Cohesion in Methods)

Название	**Название метрики и его известные синонимы**
Определение	Указывает на уровень связности между методами и атрибутами класса.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Связность объектов (Coupling Between Objects)

Название	**Название метрики и его известные синонимы**
Определение	Указывает на степень зависимости одного класса от другого.

Цели	
Процедура анализа	TODO
Обязанности	TODO

Процент публичных переменных (Percent Public Instance Variables)

Название	**Название метрики и его известные синонимы**
Определение	Определяет соотношение переменных с публичным модификатором.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Доступ к публичным данным (Access to Public Data)

Название	**Название метрики и его известные синонимы**
Определение	Показывает количество обращений к открытым или защищенным атрибутам каждого класса.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Индекс сопровождаемости (Maintainability Index)

Название	**Название метрики и его известные синонимы**
Определение	Индекс в диапазон от 0 до 100, где более высокие значения указывают на лучшую сопровождаемость/поддерживаемость.
Цели	Может применяться для оценки простоты внесения в код изменений и дополнений, легкости расширения. В целом, высокие значения характеризуют хороший, качественный низкоуровневый дизайн исходного кода.
Процедура анализа	TODO
Обязанности	TODO

Ссылочное связность (Relational Cohesion)

Название	**Название метрики и его известные синонимы**
Определение	Ссылочная связность.
Цели	Среднее количество внутренних связей на тип. Типы в сборке должны быть тесно связаны друг с другом, но не должны быть чрезмерно связаны.
Процедура анализа	TODO
Обязанности	TODO

Значимость типа (Type Rank)

Название	**Название метрики и его известные синонимы**

Определение	Значимость типа - величина того, насколько тип критичен и важен для приложения. Типы с высоким значением должны быть хорошо протестированы и спроектированы, так как ошибки в этих типах будут иметь большее влияние на приложение.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Значимость метода (Method Rank)

Название	**Название метрики и его известные синонимы**
Определение	Подобно значимости типа (Type Rank), Method Rank можно использовать для определения методов, которые имеют наибольшее значение для приложения.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Стили кодирования

Метрики соответствия исходного кода единому стандарту оформления, специфичного для выбранной платформы/фреймворка/языка, важны для упрощения поддержки и сопровождения кода. Должны быть подготовлены профили для каждой платформы. Обычно такие профили описывают правила выравнивания, именования, порядка, читаемости, документируемости кода.

TODO

Правила кодирования

Метрики соответствия правилам кодирования определяют, насколько исходный код соответствует правилам корректной работы, описанным для каждого языка или фреймворка. Например, это может быть правила освобождения ресурсов, захвата и освобождения блокировок, работы с памятью, инициализации переменных, обработки исключений и другие подобные правила, созданные на основе рекомендаций правильного и безопасного кодирования.

TODO

Дублирование кода

TODO

Правила документирования кода

TODO

Сканер уязвимостей исходного кода

Метрики уязвимости кода позволяют специалистам по информационной безопасности выявлять и подтверждать уязвимости и признаки несанкционированного доступа, а разработчикам - ускорить исправления небезопасного кода на ранних стадиях, пока это не дорого.
На самом базовом уровне тестирование приложения на предмет уязвимости направлено на то, чтобы исключить возможность неправильной работы кода, обеспечить бесперебойную работу приложения и спасти владельца от финансового и репутационного ущерба.

Есть несколько основных видов автоматического тестирования безопасности:

- Static application security testing (SAST)
- Dynamic application security testing (DAST)
- Interactive application security testing (IAST)
- Runtime application self-protection (RASP)

Использование статического тестирования безопасности приложений (SAST) позволяет обнаруживать дефекты на ранних этапах разработки. Ди намическое тестирование безопасности приложений (DAST) обеспечивает проверку API приложения до его запуска. Затем интерактивное тестирование безопасности приложений (IAST) использует специальное ПО для анализа запущенных приложений. И, наконец, самозащита приложений во время выполнения (RASP) может определять происходящую атаку и реализовывать необходимые меры. Все эти подсистемы могут генерировать специальные метрики, которые также будут учтены воротами качества при сборке и развертывании приложения.

Статическое тестирование безопасности приложений (SAST)

SAST выполняет «тестирование белого ящика», что означает тестирование внутренних структур приложения, а не его функциональности. SAST для обнаружения уязвимостей работает на том же уровне, что и исходный код. Поскольку анализ SAST проводится до компиляции кода и без его выполнения, этот инструмент можно применять на ранних этапах жизненного цикла разработки программного обеспечения.

Метрики SAST

TODO

Динамическое тестирование безопасности приложений (DAST)

DAST - это метод «черного ящика», то есть он выполняется извне, без понимания внутреннего устройства приложения. Принцип работы обычно заключается в введении ошибок для проверки путей выполнения кода в приложении. DAST не требует исходного кода или двоичных файлов, так как он анализирует, выполняя само приложение.

Метрики DAST

TODO

Интерактивное тестирование безопасности приложений (IAST)

IAST использует специальные программные средства для оценки работы приложения и выявления уязвимостей. IAST использует «агентный» подход, то есть агенты запускаются для постоянного анализа работы приложения во время автоматизированного тестирования. Основное отличие IAST от SAST и DAST заключается в том, что он работает внутри приложения. Доступ к такому широкому диапазону данных увеличивает охват IAST по сравнению с исходным кодом или сканированием HTTP-запросов.

Метрики IAST

TODO

Сканер лицензий

TODO

Архитектурные ограничения

TODO

Метрики CI

Метрики времени автоматизированной сборки приложения. Интересно время собственно сборки, а так же различные пользовательские метрики, полученные из модульных тестов (например, некоторые из них могут быть "дымовыми").

Метрики тестовых прогонов

Средний процент обнаруженных неисправностей (Average Percentage of Faults Detected)

Название	**Название метрики и его известные синонимы**
Определение	Средний процент обнаруженных сбоев (или APFD) предназначен для измерения частоты сбоев или ошибок по отношению к проценту выполняемого набора тестов (отношение числа ошибок к числу тестов в наборе тестов). Показатель является быстрым и простым способом оценки качества конкретных компонентов в приложении и позволяет сосредоточить усилия по тестированию, рефакторингу и кодированию там, где они наиболее необходимы.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Серьезность ошибок (Fault Severity)

Название	**Название метрики и его известные синонимы**
Определение	Метрика, которая рассматривает каждую ошибку, обнаруженную в течение жизненного цикла разработки и присваивает этой ошибке уровень серьезности от низкого до высокого. Качества и атрибуты, используемые для определения серьезности, будут зависеть от организации и бизнес-требований приложения. Помогает расставить приоритеты для тестирования.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Уровень тестового покрытия

Все метрики покрытия предназначены для определения соотношения затронутого запуском автоматизированных тестов кода ко всему исходному коду приложения. Обычно имеет смысл держать этот показатель достаточно высоким, но цели достичь 100% нет, так как это дорого и может сигнализировать скорее о формальном подходе к соблюдению требования высокого покрытия.

Функциональное покрытие (Functional Coverage)

Название	**Название метрики и его известные синонимы**
Определение	Предназначено для измерения количества функций, методов, классов, которые охватываются набором тестов. Хорошо подходит для тестирования реальных бизнес-требований.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Покрытие операторов (Statement Coverage)

Название	Название метрики и его известные синонимы
Определение	Полное, построчно или оператор за оператором, в зависимости от используемого языка, тестовое покрытие.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Покрытие ветвления (Branch Coverage)

Название	Название метрики и его известные синонимы
Определение	Измерение количества ветвей (то есть фактически условных операторов) в кодовой базе, которые были выполнены во время тестирования. Важно держать это значение максимально высоким.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Метрики процесса сборки

Крайне полезно измерять некоторые параметры сборки, так как они косвенно могут свидетельствовать о проблемах с дизайном кода, неудачном применении кодогенераторов или чем-то подобном, что трудно заметить самим разработчикам. Например, заметный рост времени сборки может означать излишнее количество зависимостей между модулями или неявную загрузку ресурсов периода компиляции.

TODO

Метрики CD

Метрики процесса развертывания приложения в реальном рабочем окружении. Имеет смысл замерять время выполнения отдельных операций (и анализировать динамику), так как большое время некоторых операций может сигнализировать о проблемах в инфраструктуре (медленная сеть , проблемы с пропускной способностью, нехватка ресурсов и т.п.)

TODO

Метрики управления

Бывает полезно узнать, в каком ритме работает команда программистов, какая статистика открытых/закрытых багов, объем технического долга, скорость реакции на появление багов, трассировка требований (от запланированной задачи к реализованной функциональности) и подобное.

Для реализации возможно применения плагина SonarQube Connector for Jira

https://marketplace.atlassian.com/apps/1217471/sonarqube-connector-for-jira?hosting=cloud&tab=versions

TODO

Метрики эксплуатации

Наиболее важный набор метрик порождается развернутыми в реальном окружении приложениями. Здесь может быть статистика сбоев, времени выполнения операций и запросов, уровень соответствия приложения нефункциональным требованиям.

Производственные инциденты (Production Incidents)

Название	Название метрики и его известные синонимы

Определение	Критерий измерения качества и производительности приложения, когда оно приблизится к готовности к эксплуатации или уже введено в эксплуатацию. Чрезвычайно полезно измерять производственные инциденты, как по количеству, так и по их частоте с течением времени.
Цели	
Процедура анализа	TODO
Обязанности	TODO

Шаблон метрик:

Название	**Название метрики и его известные синонимы**
Определение	*Атрибуты объектов, которые измеряются с использованием этой метрики, правила и формулы расчета метрики (в том числе из каких других метрик она может рассчитываться).*
Цели	*Список целей и вопросов, связанных с данной метрикой. Основания для сбора метрики.*
Процедура анализа	*Как предполагается использовать метрику?* *Как увидеть ее значения (как применить соответствующие инструменты)?* *Как можно интерпретировать ее значения?* *Допустимый диапазон, границы, референсные значения?* *Математические модели, применяемые для анализа?* *Процедура калибровки?* *Что еще может влиять на измерения (окружение, внешние условия)?*
Обязанности	*Кто будет собирать и обобщать данные измерений, готовить отчеты и анализировать данные?*

Примитивные метрики

Перечень примитивных метрик, которые собираются для вычисления основных метрик.

Шаблон для примитивных метрик:

Название	**Название примитивной метрики**
Определение	*Однозначное описание метрики в терминах рабочего окружения проекта.*
Процедура сбора	*Описание процедуры сбора метрик.* *Какие инструменты применяются для сбора?* *В какой момент жизненного цикла системы/программного компонента происходит сбор?* *Какая процедура проверки используется?* *Где хранятся измерения (место, формат, точность)?*
Обязанности	*Кто отвечает за сбор и анализ данных измерений?*

Приложения

Методы расчета, таблицы для оценок, подробности процедур, т.п.