О качестве кода замолвите слово

- 
- Фильм "грань будущего" - это одна из возможных аллегорий борьбы за качество промышленного(enterprise) кода. Когда решаешь одну и ту же задачу разными способами, на разных проектах, в условиях неотвратимо сжатых сроков и бюджетов.

Что это

Статья, а также набор утилит, скриптов, примеров и наставлений для анализа качества JS* кода в динамике за год. Рекомендации даны на основе опыта 5 лет работы в промышленных проектах.

- Ссылки для торопливых
  - скрипты
  - Шаблон отчёта
  - Пример анализа ENG
  - Пример анализа wallet
  - Пример анализа material UI
  - Пример анализа taiga UI
  - Пример анализа AG Grid
- Ссылки на утилиты:
  - eslint - js/ts linter with plugins: typescript, angular, promise, react
  - stylelint - css
  - cspell - spelling
  - prettier - formatter
  - jscpd - copy paste

Для чего это

Успех проекта зависит от скорости работы команды. Эта скорость определяется нашими возможностями и ограничениями. Поскольку мы, разработчики, работаем с кодом, то его сложность/качество является для нас главным объективным ограничением.

Качество кода само по себе складывается из субъективных и объективных показателей, и обычно описывается размером техдолга. Впрочем, даже такой простой и распространённый показатель как количество задач зачастую применяется редко и неточно, то есть играет незначительную роль в борьбе за живучесть проекта.

Кроме ежедневной борьбы за живучесть, анализ качества кода полезно делать перед:

- выходом проекта в open source
- планированием вех и размера команды
- принятием решения о входе в проект

Объективные показатели качества могут помочь менеджерам начать думать о качестве кода. И не только думать, но и делать процессы более управляемыми/предсказуемыми. Даже в масштабах организации, т.е. более чем в одном проекте, решая проблему "курицы и яйца": хорошие разработчики хотят работать с хорошим кодом.

Перед входом в проект важно понимать совместимость возможностей и ограничений кандидата с одной стороны, и команды/кода/инструментов/процессов с другой. В этой куче субъективных метрик проще всего замерить именно код, хотя даже его оценивают субъективно, ибо "нет времени". Команды набираются реже впритык, а чаще всего ниже минимума, что порождает авралы и переработки, не даёт возможности объективно оценить происходящее.

Чтобы понять насколько проект/код живучий/качественный, то есть тонет он или всплывает, необходимо ввести несколько дополнительных счётных показателей.

Таких потенциальных показателей очень много, и более того, не все из них легко посчитать, а посчитав, применить на практике, в рамках борьбы за живучесть.

Лёгкость счёта можно обеспечить минимализмом - однострочниками bash. Это позволит не произносить в слух страшные слова: бюджет, сроки, SP, DevOPS, CI/CD.

С лёгкостью применения немного сложнее потому что применение затрагивает чувства людей. Предотвратить споры поможет использование относительных, а не абсолютных величин, т.е. сравнение показателей сейчас и год назад, а также использование процентных соотношений: количество затронутых проблемами файлов.

Итого здесь собрана пачка утилит, примеров использования методики, однострочников и наставлений. Они позволяют оценить качество кода, и применить оценки на практике за считанные дни.

Если же всё это покажется вам слишком сложным, то местные артефакты можно применять по частям, т.к. они слабо связаны.

Если этого показалось мало, то вот вам пачка ссылок для дальнейших поисков совершенства.

- подборка утилит
- скрипт анализа частоты исправления одного и того же кода

- подборка скриптов git
- документация по правилам линтера
    - https://eslint.org/docs/rules/
    - https://typescript-eslint.io/rules/

Преимущества

- утилиты проверены на трёх типах проектов: react/angular/JS
- самый большой проект: 200к+ строк кода ts
- методика объясняет как сделать анализ полезным для разных типов проектов
- в отличии от утилит типа codecov, codescene, sonar и типовых наборов правил:
    - позволяет получать более значимые результаты. Типовые решения настроены выдавать как можно меньше типов ошибок, не подсказывают что делать дальше, и выдают слишком много мусора
    - скорость анализа слабо зависит от размера проекта. Используется одинаковая методика, скрипты практически не требуют настройки производительности
    - не требуют настройки окружения/CI,
    - работают локально, нет внешних/облачных сервисов, которые могут сломаться после обновления версий или по желанию вендора
    - набор скриптов, которые легко понять и переиспользовать по частям

Трудности

- это больше конструктор, чем готовое решение: нет пакета npm, необходимо выкачивать репозиторий с утилитами
- может быть трудно подобрать совместимые версии ESLint, его плагинов, его конфигурации, фреймворка
- скрипты не покрыты защитой от некорректных данных
- проверка дублирования кода работает только после `git commit` в папке линтера
- работа скриптов проверена только для ОС Linux, может потребоваться развернуть виртуалку или воспользоваться тестовыми стендами
- необходимо удалять вложенные конфиги eslint из всех папок в проектах с монорепами/NX
- в монорепах/NX неудобно анализировать вложенные проекты
- в логах нужны абсолютные пути к файлам, чтобы заработала навигация в IDE по клику на строку(eslint.log-->someClass.ts)
- для разных фреймворков(react/angular/JS) необходимо подбирать свои группы правил ESLint
- `jscpd` и `stylelint` обрабатывают чуть меньшее, чем скрипты количество файлов, возможно из-за настроек `ignore`, поэтому затруднительно считать соотношения с результатами скриптов
- если jscpd зависает на больших проектах необходима настройка
    - выключить blame
    - уменьшить порог срабатывания
    - проверять отдельно html|ts|css

Как это использовать

Скрипты разделены на две группы - одна для запуска в папке линтера, другая - в папке проекта для анализа. Потому что интегрировать скрипты линтера в папку проекта может быть затруднительно технически, организационно, а также из-за сопротивления членов команды.

В шаблоне отчёта указаны однострочники, их можно запускать руками.

Скрипты и утилиты также можно подключить непосредственно в IDE.

Результат работы скриптов - папка с логами. Данные из них заносятся вручную в шаблон. Поскольку групп скриптов два, то на выходе у них два главных файла: log.md и log.project.md, где сырая информация отжата и подготовлена.

исходные данные

Чтобы продемонстрировать методику в действии были проанализированы следующие проекты:

- Bitpay Wallet (formerly Copay) is a secure Bitcoin, Bitcoin Cash, Ethereum and ERC20 wallet platform for both desktop and mobile devices
  - Пример анализа wallet
- The Angular Material common UI components and tools to help you build your own custom components
  - Пример анализа material UI
- Taiga UI is fully-treeshakable Angular UI Kit 86082f2 1,906 коммитов
  - Пример анализа taiga UI
- AG Grid is a fully-featured and highly customizable JavaScript data grid ba951f2 28,150 коммитов
  - Пример анализа AG Grid

Отчёт

Почитайка aka README - пожалуй, это самый важный файл в проекте. Проект не может развиваться без программистов. А этот файл именно для них, и он неразрывно связан с кодом. Поэтому логично разместить отчёт о качестве кода именно тут. Для логов можно завести отдельную папку doc, откуда удобно прокликиваться до строчек в исходниках.

Для разделов связанных с техдолгом указана важность и сложность исправления. Для разных проектов она может меняться в зависимости от возможностей команды и потребностей бизнеса. Важность и сложность примерная, субъективная, для сферического проекта промышленной админки возрастом более 1 года.

Пробежимся по разделам шаблона, добавим наставления по их использованию.

Общие данные

Этот раздел неплохо дополняет результаты анализа, расширяя его до полноценной почитайки.

Техдолг

Исходя из возможностей/ограничений проекта/команды необходимо прикинуть в каком порядке его будут сдувать. Для более простых частей можно даже сдувать в масштабах проекта, для более сложных частей - по файлам/папкам/доменам.

Что полезно написать, чтобы не искать в чатиках:

- Ссылки на доски, вехи в жире для техдолга
- Ссылки на статьи про техдолг в базе знаний

- Краткий план снижения техдолга

Типичный сценарий:

1. берём задачки техдолга из бэклога, если нечего делать
2. переписываем код, связанный со старыми библиотеками
3. увеличиваем покрытие тестами
4. правим имена переменных/классов/методов

Прогрессивный сценарий:

1. регулярно проводим анализ состояния техдолга, согласовываем приоритеты частей техдолга, прикидываем трудозатраты
2. выделяем на регулярной основе часы/дни/недели для техдолга, например:
    - один день в спринт
    - спринт через два спринта
    - один спринт после закрытия вехи
3. Первая очередь
    - ошибки компиляции - это ошибки в проде
    - покрытие типами данных кода фронта со стороны API/BE - это облегчение любых изменений, защита от появления ошибок
    - покрытие простыми тестами инициализации модулей/компонентов - это защита от поломки модулей/компонентов целиком
    - внедряем автоматизацию анализа качества кода - проверка кода перед локальным коммитом, локальные линтеры
4. Вторая очередь
    - заменить библиотеки/велосипеды, которые больше всего снижают скорость команды. Можно начать с написания декораторов/моделей/интерфейсов для облегчения дальнейшей замены библиотек. Не падаем духом, практикуем совместное программирование в сложных случаях.
    - покрытие типами данных кода фронта: методы
    - покрытие тестами ввода/вывода методов
    - внедряем автоматизацию анализа качества кода - проверка/тестирование кода CI/CD
5. Третья очередь
    - меняем структуру файлов/папок проекта для облегчения поиска кода
    - внедряем автогенерируемые типизированные библиотеки API со стороны FE и BE
    - покрытие автотестами, интеграционными/регрессионными тестами
    - внедряем автоматизацию анализа качества кода - покрытие кода тестами/документацией
6. Четвёртая очередь
    - разбиваем код на библиотеки
    - осваиваем микрофронты на стороне FE, микросервисы на стороне BE
    - добавляем уровни абстракции в хранилище/API

Да, растущий проект рано или поздно упрётся в необходимость разделения кода между разными командами. Микрофронты/микросервисы - наиболее универсальный способ решения этой задачи. Главный недостаток подхода - необходимость увеличения команд для обслуживания новых уровней абстракций. Скорее всего, потребуется отдельная команда DevOps.

Зависимости

важность: средняя сложность исправления: средняя

Вроде бы очевидный, не требующий описания пункт. Всё же есть в `package.json`, но нет. В этом файле плохо с человеко-читаемостью, зависимости никак не сгруппированы. Это то, о чём обычно не знают рекрутёры, или очень скупо рассказывают на интервью. Кроме того, там не хватает ссылок на документацию и дату выпуска библиотеки.

Прежде чем браться за оценку задач в проекте, было бы здорово почитать обо всех значительных технологиях, которые есть в проекте. Особенно о тех, которых нет в зависимостях, т.е. о самописных велосипедах. Ну хотя бы знать где они лежат, чтобы накинуть пару SP сверху, если уж документации нет, а авторы давно ушли в закат. Идеальный вариант - взять отдельное время(часы/дни) для изучения, написания тренировочного проекта с этими технологиями.

Зависимости влияют на производительность. Например: на скорость сборки, которая зависит от размера библиотек. Размеры библиотек относительно легко собираются webpack. Однако необходимо подобрать версию webpack-bundle-analyzer под соответствующую версию, поддерживаемую фреймворком

С точки зрения пользователя старые библиотеки, скорее всего, также снижают производительность. Это самая понятная часть техдолга и для менеджера, и для программиста. Поэтому замена старых библиотек на новые - это то, что делают в первую очередь. Год выпуска ищется в интернетах так: `lodash changelog`

Больше всего разработчиков раздражает неочевидная структура связей частей кода. А самое сложное в анализе связей - это построить граф. Он строится на базе import, зачастую не полностью. Связи через директивы в шаблонах html и через маршруты URL вообще не отображаются в популярных утилитах. В angular есть ещё пара способов связывания ts файлов: через router resolve и HTTP interceptor. Внешние библиотеки типа NGRX также скрывают связи, хотя и предоставляют взамен инструменты для отладки состояний и строгие наставления для структуры файлов.

Для облегчения работы с графом я допилил одну брошенную репу. Добавил туда возможность экспорта в редактор yed. На вход подаём webpack stats.json, на выход есть разные варианты dot/json, но удобнее всего graphml, который можно покрутить в редакторе, автоматически расположить узлы и связи.

Список инструментов для анализа зависимостей:

- валидация/дедупликация зависимостей
- граф плагин vscode
- граф без webpack
- nx
- compodoc
- онлайн npm
- скриптом используя dot language и graphviz

С циклическими зависимостями немного проще, их ловят и компиляторы, и webpack.

Размер проекта

важность: средняя сложность исправления: средняя

Он влияет на всё. И его необходимо непрерывно сдувать: удалять неиспользуемый код, переиспользовать классы и функции, использовать библиотеки. Можно ставить цели по этим метрикам, например: увеличить количество файлов, снизить количество строк.

Я отдаю предпочтение количеству строк, а количество файлов использую для расчёта процентных соотношений. Поэтому в шаблоне далеко не все из собранных показателей, а только самые полезные.

Комментарии

важность: низкая сложность исправления: низкая

Это та самая документация, которой нет. Тема спорная. Чаще всего комментариев или вообще нет или очень мало. Я встречал проекты, где объём файлов целенаправленно уменьшали за счёт удаления всех комментариев. Средний вариант: минимум(желательно одна строка) текста, описываем не как работает код, а почему(для чего) он написан.

Если вы регулярно пишете комментарии, то их количество достаточно легко регулировать во время проверки кода(code review).

При желании можно допилить скрипты для расчёта покрытия функций/классов/типов.

Эксперты

Пожалуй, это самый полезный раздел. Помогает найти кому задавать вопросы по коду

Возраст проекта

Позволяет косвенно оценить сложность кода. Количественно - сколько человеколет вложено в проект.

Ошибки компиляции

важность: высокая сложность исправления: низкая

То, с чего стоит начать в первую очередь. Это ошибки, которые могут выстрелить на проде в пользователя, и их относительно легко исправить. Например, для старых angular проектов, это могут быть ошибки в html шаблонах, которые остались после переезда на новую версию. Эти ошибки можно включать в настройках сборщика опцией fullTemplateTypeCheck, которая появилась в 9 версии.

Статический анализ

важность: высокая сложность исправления: высокая

Большинство ошибок, которые отлавливают статические анализаторы - стилистические. Т.е. они про стиль кода. Именно к такому типу ошибок склоняются популярные наборы правил линтеров.

Большая длинна и сложность - самый полезный тип ошибок, которые ловят линтеры. Длинна файлов/функций, количество циклов и ветвлений сильнее всего затрудняют чтение кода.

Человекопонятные названия переменных/функций/классов можно относительно легко поддерживать во время проверки кода. А вот чтобы держать в узде сложность и длину необходимо серьёзно вчитываться в код во время проверки(code review), что зачастую затруднительно.

Покрытие типами данных - это, чаще всего, наследие старых версий фреймворка. Версии библиотек подняли, добавили тайпскрипт, а типами код так и не покрыли. Покрытие типами легче всего начинать со стороны API. Скопировать json из chrome dev tools, и конвертировать его плагинами IDE сразу в типы и объекты-заглушки. Получится пара: заглушка и модель/интерфейс. Эту пару можно сразу использовать в тестах, а также постепенно протягивать внутрь кода.

Асинхронный код - наиболее сложный для поиска и исправления тип ошибок. Они проявляют себя в сложных сочетаниях условий выполнения кода и могут зависеть от данных, окружения, действий пользователя. Самый простой путь - перенять лучшие практики из более зрелых проектов:

- https://github.com/xjamundx/eslint-plugin-promise
- https://github.com/cartant/eslint-plugin-rxjs

Стили

важность: низкая сложность исправления: низкая

Похоже, что это наименее важные ошибки, т.к они про стиль кода. Ошибки серьёзно влияющие на пользователя скриптами не ловятся. Для их отлова необходимо проводить тяжёлое регрессионное(со скриншотами) тестирование на множестве вариантов окружений(ОС+браузер).

уязвимости пакетов

важность: высокая сложность исправления: средняя

Это `npm audit`. Сейчас github создаёт автоматические PR в наши репы с исправлениями уязвимостей. Устраняется относительно легко, если повезёт не затронуть проблему совместимости версий библиотек. Если не повезёт - обновлять придётся много чего, включая основной фреймворк. В проектах с непубличным кодом проблема стоит чуть менее остро, т.к. открытый код атакуют роботы, которые знают об уязвимостях заранее.

Для усложнения задачи атакующим можно выносить бизнес-логику в тыл(backend), и делать проверку проверку входных данных.

Покрытие тестами

важность: высокая сложность исправления: высокая

Самое неприятное для устранения дело. Мало того, что задачу надо сделать побыстрее, успеть протащить её через тестирование и проверку кода, так ещё и код нужно, по-хорошему, удваивать. Поэтому если тесты и есть, то очень скудные. Хотя даже такие скудные тесты помогают ловить простые, но неочевидные ошибки, которые появились во время исправления других ошибок.

Скрипты позволяют косвенно оценивать покрытие кода тестами по количеству строк, независимо от тестового фреймворка и обходя проблемы интеграции монореп/микрофронтендов. Конфиги CI/CD, фреймворков, NX/SingleSPA править не нужно. Хотя и результат получается не такой богатый как в утилитах типа Istanbul reporter или codecov

фреймворк Angular

важность: средняя сложность исправления: низкая

Разработчики зрелых фреймворков любят мериться производительностью, поэтому скриптов и конфигов на эту тему написано очень много. Для примера наиболее простой метрики можно посчитать покрытие теми самыми настройками производительности, которые идут в комплекте с фреймворком.

Для angular - это onPush.

Более сложные для поиска и исправления метрики производительности требуют организовывать нагрузочное тестирование, и использовать специальные инструменты.

В Angular есть хороший набор наставлений по организации кода, в частности - про именование файлов. Так их проще искать. В слабо структурированных проектах можно замерять покрытие стандартизованными именами файлов

Дублирование кода

важность: низкая сложность исправления: средняя

Для ангуляр проектов он может указать на общую проблему шаблонного подхода в html файлах, который "исторически сложился" со времён первой версии или после перехода с не менее старого фреймворка. Этот случай наиболее тяжелый потому что необходимо переносить логику построения html шаблонов в директивы и ts код.

В целом, переиспользование кода балансирует между универсальными и специальными решениями. Меньше кода - меньше ошибок, и больше тестов.

Про плюсы/минусы дублирования кода можно почитать в концепциях DRY/WET

Правописание

важность: низкая сложность исправления: низкая

Наиболее лёгкий для исправления тип ошибок. Отсутствие ошибок правописания увеличивает скорость поиска информации в коде.