Практика по настройке репозитория

Юрий Литвинов
y.litvinov@spbu.ru

На этой практике мы разберём чеклист по правильному оформлению репозитория, принятый на кафедре системного программирования.

Вот сам челист, как он есть:

1	Лицензия	
2	Лицензия правильно применяется к репозиторию	
3	Используемые третьесторонние компоненты и материалы совместимы с лицензией	
4	Настроенный CI	
5	Модульные тесты в CI	
6	Линтер в CI	
7	В репозитории нет результатов сборки, настроен .gitignore	
8	В репозитории нет секретной информации (паролей, ключей и т.п.)	
9	Различные сторонние анализаторы (если уместно)	
10	Для кода на C/C++: использование санитайзеров	
11	README.md, плашки CI и анализаторов	
12	README.md, общее описание проекта	
13	README.md, пример использования	
14	README.md, инструкция по сборке и запуску	
15	README.md, как помочь проекту (если уместно)	
16	README.md, корректные названия компаний и организаций	
17	Настроена секция About, указаны темы (topics)	
18	Код соответствует принятому в сообществе стилю кодирования	
19	Имеется техническая документация (в README.md или на вики)	
20	В коде достаточно комментариев	
21	Комментарии к коммитам адекватны, коммиты показывают историю проекта	
22	В главной ветке адекватная история коммитов	
23	Добавление релизов	
24	Настройки репозитория	

Вот некоторые пояснения с комментарием касательно того, как это относится к нашему курсу.

1. Отдавайте предпочтение разрешающим лицензиям. Код мы рекомендуем лицензировать под Apache License 2.0, MIT License, BSD 3-Clause License.

    - *Лицензия у нас обязательна, да. Выбирайте любую — и в корень репозитория.*

2. Каждая лицензия имеет требования к тому, как её правильно применить к файлам в репозитории. Например, Apache License 2.0 позволяет себя применять пофайлово, для чего требует включения в лицензируемые файлы стандартного заголовка. Также распространено использование файла LICENSE в корне репозитория и ссылка на него в заголовке каждого файла. Поищите для своей лицензии, как её правильно применять.

    - *StyleCop требует наличие шапки с лицензией, хотя бы в виде ссылки на полный текст. Погуглите, как для вашей любимой лицензии правильно, и сделайте так.*

3. Если используете чужую интеллектуальную собственность, найдите на неё лицензию и проверьте, что вы действительно выполняете её требования (например, проект, лицензированный под Apache License 2.0 *не может* использовать код, лицензированный под GPL v2). Если чужой материал не имеет лицензии (например, просто картинка из интернета или кусок кода со Stack Overflow), использовать его *нельзя*.

    - *Думаю, что в репо с домашками проблем с лицензионной чистотой быть не должно, но погуглите, под какой лицензией сам .NET и все библиотеки, что вы используете (тот же NUnit).*

4. Если вы используете GitHub, Continuous Integration-систему удобнее всего настраивать на GitHub Actions, однако вполне допустимы и сторонние системы, такие как AppVeyor, CircleCI. Если вы используете компилируемые языки, CI-система должна проверять собирабельность кода в каждой ветке репозитория и при пуллреквесте. Если интерпретируемые, проверять качество кода и работоспособность.

    - *GitHub Actions, однозначно.*

5. В проекте должны быть модульные тесты (за редкими исключениями, где они неприменимы или бессмысленны), и модульные тесты должны запускаться в CI.

    - *dotnet test выполнит и сборку, и запуск тестов, поэтому всё можно сделать одной командой. Но надо для каждого решения в репозитории, у нас их, скорее всего, несколько.*

6. Должен быть настроен форматтер/линтер, следящий за качеством кода, и также запускаться в CI. Если линтер выдаёт ошибки, сборка должна не проходить.

    - Например, для F# линтер — это FSharpLint, для Python — flake8, форматтер для F# — это Fantomas, для Python — black. Нет ничего плохого в том, чтобы использовать форматтер и линтер в CI, настроенные так, чтобы некачественный код даже не доходил до фазы сборки.
    - Запуск линтера может быть отдельной задачей в CI, чтобы не гонять его по нескольку раз в разных конфигурациях сборки.
    - Имеет смысл сделать запуск линтера локальным pre-commit hook в git, чтобы некорректный код даже не позволяли закоммитить. Если есть возможность интегрировать среду разработки и линтер/форматтер, сделайте это. Например, Visual Studio Code легко подружить с Fantomas, чтобы он запускался при каждом сохранении файла и делал как надо.

- *StyleCop при сборке запускается сам, если добавлен в зависимости в проекте, так что линтер вручную можно не настраивать. Но надо, чтобы StyleCop был в проекте.*

7. На GitHub файл .gitignore можно выбрать при создании репозитория, но также часто требуется ручная модификация. Должно быть так, чтобы все файлы, которые .gitignore позволяет закоммитить, реально нужно было коммитить. *В репозитории не должно быть результатов сборки,* (то есть папок bin, obj, pycache и т.п.), в идеале не должно быть бинарных файлов вовсе (только если очень надо и вы реально знаете, что делаете).

   - *.gitignore обязателен.*

8. Разумеется, в репозитории (включая историю коммитов) не должно быть ничего, что вы не хотели бы публиковать (например, ключей авторизации от сообществ ВКонтакте). Если пользуетесь GitHub, кое-что он умеет ловить сам, для этого надо убедиться, что в «Settings/Code security and analysis» включено «Push protection». Но, разумеется, большую часть секретов он не найдёт.

   - *Думаю, ни у кого не может быть такой проблемы, но вдруг...*

9. Используйте сторонние анализаторы для слежения за качеством кода: например, CodeCov для анализа тестового покрытия, CodeFactor или Codacy как продвинутый статический анализатор. Чем больше инструментов следят за тем, что всё хорошо, тем лучше.

   - *CodeCov попробуйте настроить, если останется время.*

10. Языки типа C и C++ дают возможности для работы на низком уровне, но благодаря этому повышается вероятность появления таких ошибок, как небезопасная работа с памятью или undefined behavior. Поэтому для проектов на этих языках стоит включать санитайзеры при сборке, тестировать санитайзерами на CI, а также запускать инструменты для отслеживания утечек памяти (например, Valgrind).

    - *Не наш случай.*

11. Добавьте в README.md плашки CI и анализаторов (штучки, на которых написано «CI passing» или что-то такое). В документации конкретной CI-системы или анализатора обычно легко найти, как добавить плашку в Markdown. Это поможет посетителям сразу посмотреть статус кода.

    - *Плашка со статусом CI обязательна. Успеете CodeCov — его плашку тоже можно добавить.*

12. Напишите в README.md пару абзацев текста, про что вообще проект. Помните, что код вы пишете не только для себя, в ваш репозиторий придут люди, которые вообще не имеют идей, о чём это.

    - *Это обязательно даже для репо с домашкой. Напишите, что это репо с домашкой по такому-то курсу.*

13. Опишите типичный пример использования, если уместно, с картинками или gif-ками. Включая информацию, откуда брать датасеты, куда подкладывать конфигурацию и т.п., чтобы любой пользователь мог с чистого листа запустить проект и понять, что у него получилось.

    - *Напишите, как собрать и запустить любую домашку. У нас — довольно тривиально, но помните, что не все имеют идеи, что такое .NET, а вашим LZW могут захотеть пользоваться.*

14. Опишите также действия по сборке и внешние зависимости (версию используемых SDK и т.п.). Это всё есть в CI, но в README это всё должно быть в удобной человекочитаемой форме и заодно приводить к развёртыванию окружения, пригодного для работы над проектом (тогда как сборка в CI может быть весьма хитрой, использовать несколько Docker-образов и т.п.).

    - *Да, напишите версию .NET, может, ещё до описания процесса сборки и запуска.*

15. Если проект предполагает возможность стороннего участия (то есть имеет хоть один шанс стать знаменитым), опишите, как сторонний человек может вам помочь:

    - куда и как писать баги;
    - как связаться с разработчиками;
    - как контрибьютить;
    - где посмотреть техническую документацию и найти первый вводный баг, который можно поправить.

    - *Думаю, что неактуально.*

16. Названия каких-либо организаций, используемых инструментов или технологий должны быть написаны так же, как в официальных источниках (пример: TRIK Studio, а не Trik Studio, RISC-V, а не RISC-5).

    - *Думаю, что тоже не очень актуально.*

17. Оформите секцию About: стоит добавить подходящие темы (topics), чтобы ваш репозиторий было легче найти, и описание (description), чтобы стороннему человеку было понятно, зачем репозиторий нужен (кратко, одним предложением — подробное описание в README.md).

    - *А вот это надо сделать.*

18. Проверьте, что код в репозитории адекватно оформлен. Если на Python, то PEP-8, если на C++, то в соответствии с Core Guidelines и т.п. — у каждого языка и даже у некоторых фреймворков есть свой стиль кодирования, проверьте, что код его уважает. Если в проекте используется свой стиль кодирования, он должен быть явно задокументирован и весь код должен ему соответствовать.

    - *Я очень надеюсь, что у вас всё хорошо по умолчанию. А если нет, StyleCop об это скажет.*

19. Где-то должно быть некое техническое описание проекта — из каких компонентов он состоит, кто за что отвечает. В идеале — полноценная архитектурная документация в виде страниц на вики, с UML-диаграммами, но если сил нет, можно ограничиться разделом в README, где кратко словами всё описать. Также вместе с/вместо вики может быть уместна документация на Read The Docs.

    - *Это пока не надо. Проекты мелковаты.*

20. В коде должны быть комментарии (в принятом для языка формате — DocString, Doxygen, Javadoc, XML Documentation и т.п.), хотя бы у ключевых классов/интерфейсов/модулей, кратко описывающие, что вообще делает класс. В идеале — для всего, что public, с документированием предположений о входных данных, инвариантов, бросаемых исключений и свойств потокобезопасности (reentrant, thread-safe и т.п.), но насколько сил хватит.

    - В идеале — по комментариям в коде должна автоматически генерироваться документация и выкладываться на GitHub Pages или тот же Read The Docs (в т.ч. как действие при сборке в CI, то есть полностью автоматически). Например, Для Python есть инструмент Sphinx, который в этом помогает.

    - *Комментарии должны быть, генерацию документации — если только за плюс балл.*

21. Комментарии к коммитам пост-фактум исправить тяжело, поэтому за ними надо следить изначально. Рекомендуется следовать соглашению Conventional Commits. Коммиты не должны быть сделаны в последний день, а должны показывать, как шла работа, от создания пустого проекта до последнего релиза. Фразы вида «я тут локально разрабатывал, потом выложил, как получилось что-то разумное» очень сильно огорчают комиссию. Могут помочь инструменты типа Mergeable, Mergify.

    - *Про Conventional Commits почитайте, если ещё не.*

22. Почистите главную ветку репозитория от лишнего. Если у вас в репозитории больше пяти маловменяемых комментариев (типа «fix») подряд, либо несколько коммитов отвечают за небольшие изменения одной и той же функциональности, лучше либо сделайте squash и склейте коммиты в один, либо измените всю историю через git rebase -i.

    - *Вот это обязательно.*

23. Если ваш проект позиционируется как библиотека, игра или какой-либо независимый инструмент, рекомендуется публиковать новые версии в качестве релизов, т.к. они предоставляют пользователям удобный способ доступа к конкретным версиям вашего проекта и информации о том, что именно изменено или добавлено в каждой версии. Рекомендуется ознакомиться со стандартами оформления релизов и Semantic versioning.

    - *Давайте для тренировки сделаем один релиз.*

24. Рекомендуется настроить правила защиты веток (например, запретить force push в main), а также инструменты безопасности и анализа кода (обновление зависимостей через Dependabot, инструмент CodeQL для автоматического обнаружения распространенных уязвимостей и ошибок в коде).

    - *force push давайте запретим.*