# **Системы документирования исходного кода**

# **Введение**

Вероятнее всего, каждый из нас сталкивался с результатами работы различных генераторов документации. Общий принцип их работы, следующий: на вход такого генератора поступает специальным образом комментированный исходный код, а иногда и другие компоненты программы, а на выходе создаётся готовая документация для распространения и использования.

Многие программисты не любят это дело и пытаются всячески избежать этой работы всеми доступными способами. Отсутствие документации кода приводит к плохой его читаемости и к сложному техническому обслуживанию для других членов команды.

Документация кода отличается от проектной документации, так как она в основном фокусируется на том, как работает система. Несмотря на то, есть несколько причин для написания документации, многие программисты, как правило, игнорируют это. Ниже я собрал ряд причин, по которым, все таки стоит писать документацию для своего кода:

* Ваш код будет поддерживаться и использоваться другими программистами в команде? Обслуживание кода становится большой проблемой, если он не был должным образом документирован.
* Вы хотите, чтобы другие программисты помогли вам, например, через открытый исходный код. Если вы планируете начать большую, коллективную работу, стоит начать документирование кода прямо сейчас!
* Документирование вашего кода делает логику гораздо более ясной, а также делает ваш код лучше.
* Недокументированный код тяжело не только передавать на сопровождение, а также порой тяжело сопровождать и самому

Даже с учетом всех вышеперечисленных преимуществ, документирование, в целом, является трудоемким процессом. Для того, чтобы обеспечить более быстрый процесс подготовки документации и последовательности стилей, вам стоит использовать специальные инструменты.

Эти инструменты помогут вам с документированием кода и в целом помогут стать на ступеньку выше. Давайте начнем!

1. Doxygen

Doxygen является отличным инструментом для генерации документации из исходного кода. Инструмент нацелен на документирование программного обеспечения, написанного на языке C++, однако на самом деле данная система поддерживает гораздо большое число других языков, таких как: C, Objective-C, C#, PHP, Java, Python, IDL, Fortran, VHDL, Tcl, и частично D. С помощью Doxygen, вы можете создать онлайн HTML документацию. Doxygen — консольная программа в духе классической Unix. Она работает подобно компилятору, анализируя исходные тексты и создавая документацию.

Самым большим преимуществом использования Doxygen является то, что вы будете иметь последовательность всей документации исходного кода. Она также может помочь вам создавать структуру кода с использованием недокументированных исходных файлов. Все, что вам нужно сделать, это настроить его соответствующим образом.

### Sphinx

Sphinx это популярный инструмент позволяющий создавать текстовые документы и преобразовывать их в различные форматы. Это удобно при использовании систем управления версиями, предназначенных для отслеживания изменений. Он доступен по лицензии BSD и поддерживает несколько языков программирования, таких как Python, C и C++. Он может быть использован как для проектной документации, так и для документации кода. Sphinx избавляет от рутинных действий и предлагает автоматическую функциональность для решения типовых проблем, например, индексирования заголовков и специального выделения кода (например, при включении в документ фрагментов кода) с соответствующим выделением синтаксиса.

### Pandoc

Pandoc не похож на другие инструменты для документации. Он действует как швейцарский нож и позволяет разработчику быстро конвертировать разметку из одного формата в другой. Pandoc имеет широкий спектр поддержки документов, в том числе textile, reStrcuturedText, LaTex, EPUB и т.д.

Кроме того, он предлагает несколько расширений синтаксиса разметки, в том числе списки определений, таблицы, сноски и т.д.

### Dr. Explain

Frontend разработка также, в определенной степени, требует документирования. Создавать документацию как для обычных, так и онлайн-приложений, написанных на любом языке программирования, в любой среде разработки, с применением любого фреймворка поможет DR. EXPLAIN. Он фильтрует ключевые элементы интерфейса, а затем извлекает связанные с ним мета данные. После этого, вы можете изменить полученную информацию, чтобы быстро создать документацию интерфейса.

### LaTex

LaTex является де-факто стандартом для документирования научных проектов. Тем не менее, он также может быть использован для других типов проектов, в том числе кода и проектной документации. Готовя свой документ, автор указывает логическую структуру текста (разбивая его на *главы*, *разделы*, *таблицы*, *изображения*) и позволяет LaTeX’у заботиться о том, как изобразить его.

### Markdown

Markdown, творение Джона Грубера, очень простой и изящный *синтаксис разметки текста*, который поможет вам писать качественный код и документации. С технической точки зрения Markdown является инструментом преобразования текста в HTML для веб-писателей, но в равной степени он может быть использован и для документирования. Как разработчик, вы можете написать документацию в Markdown, а затем использовать Pandoc, чтобы преобразовать его в любой формат, который вам нужен.

### GhostDoc

GhostDoc это расширение для Visual Studio, с помощью которого вы можете легко генерировать комментарии документа XML. Инструмент генерирует комментарии на основе нескольких факторов, в том числе имя, параметры, контекстную информацию, типы и т.д.

### Natural Docs

Natural Docs это еще один инструмент с открытым исходным кодом, который работает со многими языками программирования. Он поможет вам автоматизировать генерацию документации кода и преобразовать его в формат HTML. В настоящее время NATURAL DOCS поддерживает 19 языков программирования, среди них Python, C ++, PL / SQL, Actionscript и т.д.

### PHPDocumentor

Если вы PHP разработчик и хотите сгенерировать документацию кода из исходного кода, стоит рассмотреть PhpDocumentor. В основе работы системы лежит парсинг логической структуры PHP кода (классы, функции, переменные, константы) и привязка к ней комментариев, написанных по определенным стандартам. Инструмент также может помочь вам генерировать отчеты и графики и повысить общее качество кода.

### Liveedu.TV

Как стриминговая платформа может помочь в документировании кода? Вы можете транслировать или хранить проектную работу непосредственно на Livecoding. Вы будете иметь возможность легко открыть доступ к важным разделам проекта для членов вашей команды. Несколько преимуществ использования видео в качестве инструмента для документирования кода. Некоторые из них приведены ниже:

* Нет писанины, но есть лучшее понимание контекста.
* Agile команды могут легко отслеживать изменения в проекте.
* Технические писатели могут использовать видео документацию, дабы понять проект лучше.
* Разработчики могут вкладывать сэкономленное время в реализацию других функциональностей проекта.

# **Пять стратегий документирования кода**

## **Подход 1: размещаем “Как?” во внутренние комментарии, “Почему?” - во внешней документации.**

После разделения, пояснения *Как?* встраиваются в код в виде комментариев (примерно на каждые 5-10 строк). Пояснения *Почему?* встраиваются во внешних разделах до или после кода. Разумеется, в пояснениях *Как?* можно получить подробную информацию о *Почему?* причинах, и наоборот, поэтому разделение этих двух вопросов на практике может быть не таким простым. Но такая общая закономерность, скорее всего, верна.

Кроме того, такой подход интересен тем, что он побуждает подумать над вопросом *Почему?*. Техническим писателям, документирующим код (который пишут другие), не всегда получается учитывать *Почему?*.Иногда бывает трудно понять, какие вопросы *Почему?* существуют. Зачем использовать этот класс вместо другого? Почему так, а не так? Часто существует много разных способов достижения цели, так почему же идти по этому пути, а не каким-то другим?

В общении с разработчиками о примерах кода, стоит задавать это вопрос: *Почему?*

* существуют другие подходы, от которых они отказались? Почему?
* могут ли пользователи реализовать какой-либо другой подход, или мы специально хотим, чтобы они следовали только одному методу? Почему?
* почему используется именно этот язык / инструмент / рамки / библиотеки, а не какой-либо другой?

Что касается вставки встроенных комментариев, best practice - это лучше всего вставлять короткие комментарии на каждые 5-10 строк кода. Получается не так много комментариев, что сделать код нечитаемым, но не так мало, чтобы не объяснять, что происходит.

Также стоит обратить внимание, что комментарии встроенного кода могут быть несколько спорными. Если просто объяснять, что делает код, это может оказаться лишним для знатока языка, на котором написан код. Некоторые разработчики считают, что простой код документирует сам себя - его значение понятно тем, кто знает язык, без необходимости объяснения. Но это утверждение про опытных пользователей и не распространяется на сложный код.

## **Подход 2: сопоставление комментариев в третьей колонке**

Лучшие практики для документации в целом (не только для документирования кода) включают размещение полезных инструкций рядом с областью, вызывающей затруднения, что в контексте документирования кода может означать добавление встроенных комментариев, пронизывающих весь код. Но предположим, что нужен большой комментарий о том, что происходит в коде (поскольку уровень сложности не может быть передан в коротком комментарии). Как сопоставить длинную концептуальную / пояснительную информацию с кодом?

Если ваш комментарий больше кода, есть риск сделать код нечитаемым. Если разместить комментарий в разделах, которые идут задолго до или после кода, есть риск создать пропасть между объяснением и кодом, так что читатели не будут знать, к каким частям кода относится объяснение.

Одним из решений такой проблемы размещения документации кода является создание дополнительного третьего столбца в макете. В среднем столбце находится концептуальное объяснение, а в правом столбце - код. Получается, код и его описание сопоставляются таким образом, что читатели могут взглянуть на код, читая концептуальные объяснения. Третий столбец поддерживает необходимый контекст между кодом и объяснениями.

Одной из проблем такого подхода является пространство экрана. Чтобы добавить третью колонку, нужно занять весь экран без полей.

Таким образом, код виден только частично. Код можно растянуть по горизонтали с возможностью прокрутки вправо, но, несомненно, дизайнерам тяжело дался пользовательский интерфейс, который включает в себя горизонтальную и вертикальную прокрутку.

Вдобавок, реализация размытия фокуса области экрана с кодом, на описании которого находится пользователь, должна быть технически сложной и громоздкой.

Другая проблема заключается в том, что код и пояснения к нему редко можно выровнять линейно. Предположим, есть метод в коде, который занимает всего одну строку, но описание этого метода занимает три длинных параграфа концептуального объяснения. К тому времени, когда пользователь достигает нижней части концептуального объяснения, код уже не виден. Пользователю приходится сделать много лишних манипуляций, чтобы найти соответствующий код. Дизайн пользовательского интерфейса для размещения всех этих движущихся частей не только кажется сложным, но и увеличивает нагрузку на пользователя.

Еще одна проблема, связанная с таким дизайном, заключается в том, что код часто относится к нескольким файлам. Отображение в колонке не указывает, отображается ли весь код в одном и том же файле Java или мы видим код из нескольких файлов Java. Представление вкладок в столбце кода требует еще более сложного формата документации. Сложно представить, что все это возможно сделать с помощью синтаксиса Markdown.

## **Подход 3: подход Лего**

Другой подход заключается в создании кода с нуля, уровня за уровнем, который можно назвать “подходом Лего”.

Подход Лего можно начать с пустого файла. Затем с каждым шагом добавлять все больше и больше кода, пока не получится полностью рабочий пример. Как и в случае с конструктором Лего - начинаем с основания, а затем добавляем блок за блоком, пока не получим желаемый результат.

Проблема с подходом Лего заключается в том, что технический писатель вряд ли воссоздаст логику построения, которой следовал разработчик. Скорее всего, разработчики просто отправят готовый кусок кода в документ, и тогда придется получать пояснения от разработчика о коде.

Поскольку код нелинейный, пошаговая работа с пояснениями кода не обязательно будет соответствовать подходу Лего. Код, отображаемый в верхней части, может показаться разработчикам вишенкой на торте - например, абстрагирование более сложных строк в переменные, которые используются для уменьшения сложности работы кода. Готовый код часто имеет логику, которая абстрагируется в переменные или другие ссылочные функции, так что определенные части конечного кода остаются чище и более краткими, а другие части становятся более непрозрачными. Готовый код часто слишком запутанный и сложный, чтобы документировать его любым доступным для обучения способом.

Несмотря на проблемы с подходом Лего, если нужно научить кого-то, как понимать код, нужно начать с простого и идти шаг за шагом от простого к сложному. Следующая техника объясняет такой подход от простого к сложному при помощи метафоры наутилуса.

## **Подход 4: подход Наутилус**

Подход Наутилус аналогичен подходу Лего, но вместо того, чтобы описывать куски работы, которые выполняются один за другим в порядке сборки, здесь описываются основные простые шаблоны, которые должны знать пользователи. Начинается с самого простого кода, а затем проект увеличиваться и увеличиваться по мере необходимости, как растущая спиральная структура оболочки наутилуса.

Пол Густафсон, управляющий компанией, занимающейся техническим обслуживанием письма, в Bay Area, которая называется экспертная поддержка, представляет подход с метафорой Наутилуса, описанный здесь. Пол говорит, что наутилус является хорошей метафорой для технической коммуникации, потому что наутилус представляет спиральную схему (последовательность Фибоначчи), которая начинает с малого и постепенно растет по мере необходимости.

Пол пишет:

«Содействие пониманию, которое является основной технической связью, происходит наиболее эффективно, следуя аналогичной схеме….  
Когда ваше понимание невелико, вы учитесь лучше, когда первый урок дает информацию для небольшой, простой задачи с чертами, которые сродни первым камерам наутилуса.  
… Хорошая новость для наутилуса заключается в том, что маленькие камеры следуют тому же основному плану, что и большие камеры. Если первые задания, которые освоил учащийся, в основном аналогичны более сложным задачам в учебной программе, учащийся начинает понимать и применять эти шаблоны. Чем раньше новички научатся «думать о вещах правильно», тем быстрее они «поймут», а это именно то, чего хотят и преподаватель, и ученик. (Уроки головоногих)»

Следуя подходу Наутилус, мы начинаем с простых базовых шаблонов, а затем постепенно наращиваем объем кода вокруг него по мере необходимости. Мы не начинаем с описания сложной законченной работы с самого начала. Законченная работа, вероятно, включает в себя все виды абстракций кода и перестановки для чистого, готового продукта.

Проблема в том, что мы часто хотим объяснить, как работает готовый код, проводя пользователя через все от начала до конца. У нас может быть 500 строк кода, которые мы хотим сформулировать, но подход Nautilus заставил бы нас объяснить только несколько небольших фрагментов этого кода (по крайней мере, для начала). Следовательно, существует проблема типа A-против-Z: мы описываем A (самый простой шаблон ядра), но конечным продуктом является Z. Как именно мы добираемся от A до Z? Как мы придем от простых шаблонов, которые могут занимать 20 строк кода, до чудовищно сложного, законченного кода, который занимает 500 строк?

Для технического писателя, смотрящего на окончательный код, нет ясного понимания того, как разработчик попал туда. Мы часто не можем отделить подобную логику, с которой начинал разработчик, что привело его к этой более сложной цели. Все, что мы видим, это этот сложный конец. Как декомпилировать код, чтобы восстановить логику, с которой разработчик начал? Как понять, что это были за начальные паттерны, которые положили начало всему процессу? Если вы не разрабатывали код и не являетесь разработчиком, то будет практически невозможно восстановить шаблон Наутилуса, стоящий за кодом.

В качестве другой аналогии можно рассмотреть возможность научить рисованию. Предположим, задача - описать готовую картину потенциальному художнику. Вам необходимо описать, как рисовать Мону Лизу.

Чтобы задокументировать процесс создания такой картины, что лучше: начать с верха и идти к низу? Нет, это было бы смешно. Скорее всего, лучше начинать с создания овала головы. Потом, несколько общих набросков для глаз и так далее. Может быть, набросать линии перспективы и другую основную структуру. Никто сразу не приступает к цветам, освещению и теням, верно? То же самое с кодом. Начинаем с основы, а затем прокладываем путь к большему количеству завершающих деталей.

Однако, если вы не художник, как бы вы узнали, как описать процесс создания картины? Вам нужно знать логику художника - с чего начать и как идти к концу. Если вместо этого вы начинаете с конца и пытаетесь вернуться назад, руководство по рисованию, вероятно, будет безумно сложным.

## **Подход 5: Интерактивный браузер**

Изучение основных шаблонов смещает документацию в область руководства. С этим типом обучения связан интерактивный браузер, в котором действие сочетается с обучением. Интерфейсы, выполняемые браузером, имеют своей целью помочь пользователям лучше понять результаты входных и выходных данных, чтобы пользователи могли сами убедиться в том, как работает код, используя более практичный для себя подход.

Наиболее распространенный пример интерактивной документации для API - это пользовательский интерфейс Swagger.

Swagger предлагает оригинальное сочетание документации с активным взаимодействием, которые помогают пользователям изучать API (читая и выполняя). Но выполнение запросов с помощью конечных точек API REST обычно довольно простое. Более интенсивные руководства по коду будет сложнее сделать интерактивными в браузере. Тем не менее, некоторые «Notebooks» (как их часто называют) позволяют запускать код, в частности Jupyter Notebook. Описание Jupyter:

Jupyter Notebook - это веб-приложение с открытым исходным кодом, которое позволяет вам создавать и обмениваться документами, которые содержат живой код, уравнения, визуализации и текст повествования. Использование включает в себя: очистку и преобразование данных, численное моделирование, статистическое моделирование, визуализацию данных, машинное обучение и многое другое.

У Google есть несколько вариантов блокнота для совместной работы с документацией TensorFlow, в которой есть операции, которые можно выполнять на веб-страницах.

Хотя интерактивные записные книжки выглядят круто, они кажутся огромной работой для чего-то, что может быть легко достигнуто с помощью образца приложения. Вместо того, чтобы выяснить, как вы можете скомпилировать код Python или другой язык в браузере, почему бы просто не предоставить пример приложения, которое пользователи могут загрузить, а затем выполнить локально, используя свои собственные инструменты компиляции и настройки?

Конечно, пользователям может потребоваться установить на своем компьютере некоторые утилиты и платформы, чтобы запустить примеры приложений, а также IDE, но выполнение кода в браузере может информировать не полностью пользователей обо всех необходимых настройках и подготовке, которые в конечном итоге будут быть обязательными для них, чтобы заставить код работать локально. Браузеры имеют тенденцию быть несколько жесткими в том, что они могут сделать - у пользователей может быть больше свободы экспериментировать (и учиться на этих экспериментах) на примерах приложения.