МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Гжельский государственный университет»** (ГГУ)

Колледж ГГУ

Специальность 09.02.07 Информационные системы и программирования

**РЕФЕРАТ**

**по дисциплине «Моделирование и анализ программного обеспечения»**

**на тему «Цели, задачи, этапы и объекты ревьюирования.  
Планирование ревьюирования.»**

ВЫПОЛНИЛ:

Студент группы ИСП-О-17

Шашков И.С.

ПРОВЕРИЛА:

Прокуронова А.Ю.

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

п. Электроизолятор

2019 г.

**Стандарт код-ревью**

Основная цель код-ревью заключается в том, чтобы гарантировать постоянное улучшение кодовой базы Google. Все инструменты и процессы посвящены этой цели.  
  
Здесь необходим ряд компромиссов.  
  
Во-первых, разработчики должны быть в состоянии успешно *решать свои задачи*. Если вы никогда не отправляете код, то и кодовая база никогда не улучшится. Кроме того, если рецензент сильно затрудняет *любую* работу, то в будущем разработчики не заинтересованы предлагать улучшения.  
  
С другой стороны, обязанность рецензента убедиться, что качество CL не снизит общее качество кодовой базы со временем. Это может быть сложно, потому что часто деградация происходит из-за небольшого снижения качества кода со временем, особенно если команда находится под сильным давлением сроков и чувствует, что имеет право на увеличение технического долга.  
  
Кроме того, рецензент несёт ответственность за рецензируемый код. Он хочет убедиться, что кодовая база остаётся последовательной, поддерживаемой и соответствует всему остальному, что упомянуто в разделе [«Что проверять в коде»](https://habr.com/ru/post/467039/#2).  
  
Таким образом, мы получаем следующее правило в качестве стандарта для код-ревью:  
  
**Обычно рецензенты должны одобрить CL, как только он достигает состояния, когда определённо улучшает общее качество кода системы, даже если CL не идеален.**  
  
Это *главный* среди всех принципов код-ревью.  
  
Конечно, у него есть ограничения. Например, если CL добавляет функцию, которую рецензент не хочет видеть в системе, то рецензент, безусловно, может отказать в коммите, даже если код хорошего качества.  
  
Ключевым моментом здесь является то, что не бывает «идеального» кода — бывает только код *получше*. Рецензент не должен требовать от автора полировать каждый крошечный фрагментик. Скорее, рецензент должен сбалансировать необходимость дальнейшего прогресса по сравнению с важностью предлагаемых изменений. Вместо того, чтобы стремиться к идеалу, рецензент должен стремиться к *непрерывному улучшению*. Коммит, который в целом улучшает ремонтопригодность, читаемость и понятность системы, нельзя задерживать на дни или недели, потому что он не «идеален».  
  
Рецензенты *всегда* могут оставлять любые комментарии по улучшению кода, но не очень важные изменения следует помечать, например, префиксом *Nit:*, чтобы автор знал, что это просто точка зрения, которую он может игнорировать.  
  
Примечание. Ничто в этом документе не оправдывает CL, которые определённо *ухудшают* общее качество кода системы. Такое возможно только в [чрезвычайной ситуации](https://google.github.io/eng-practices/review/emergencies.html).

**Менторинг**

Код-ревью может быть важно ещё и для обучения разработчиков чему-то новому о языке, структуре или общих принципах проектирования ПО. Всегда приятно оставлять комментарии, которые помогают разработчику узнать что-то новое. Обмен знаниями вносит свой вклад в улучшение кода системы с течением времени. Просто имейте в виду, что если оставляете чисто образовательный комментарий, не критичный для соответствия описанным здесь стандартам, добавьте к нему префикс *Nit:* или иным образом укажите, что автор не обязан его разрешать.

**Принципы**

* Технические факты и данные перевешивают мнения и личные предпочтения.
* В вопросах стиля абсолютным авторитетом является [руководство по стилю](http://google.github.io/styleguide/). Любая чисто стилевая деталь (пробел и др.), что не входит в руководство по стилю, является вопросом личных предпочтений. Стиль должен соответствовать тому, что есть. Если нет предыдущего стиля, примите авторский.
* **Аспекты программного дизайна практически никогда не проблема чисто стиля или личных предпочтений.** Они основаны на основополагающих принципах и должны определяться по этим принципам, а не просто на личном мнении. Иногда есть несколько допустимых вариантов. Если автор может продемонстрировать (либо с помощью данных, либо на основе твёрдых инженерных принципов), что определённые подходы одинаково эффективны, рецензент должен принять предпочтение автора. В противном случае выбор диктуется стандартными принципами разработки.
* Если никакое другое правило не применимо, то рецензент может попросить автора соблюдать единообразие с текущей кодовой базой, если это не ухудшает общее состояние системы.

**Разрешение конфликтов**

В любом конфликте первым шагом всегда должно быть стремление разработчика и рецензента прийти к консенсусу, основанному на содержании этого документа и других документов в [Руководстве автора CL](https://google.github.io/eng-practices/review/developer/) и этом [Руководстве рецензента](https://google.github.io/eng-practices/review/reviewer/).  
  
Когда прийти к консенсусу особенно трудно, может помочь личная встреча или видеоконференция между рецензентом и автором (если вы это сделаете, обязательно запишите результаты обсуждения в комментарии к коммиту для будущих читателей).  
  
Если это не решит ситуацию, то наиболее распространённый способ — эскалация. Часто она заключается в более широком обсуждении с командой, привлечении тимлида, обращении к мейнтейнеру или к менеджеру по разработке. Не позволяйте коммиту задерживаться из-за того, что автор и рецензент не могут прийти к соглашению.

**Что проверять в коде**

Примечание. При рассмотрении каждого из этих пунктов обязательно учитывайте [Стандарт код-ревью](https://habr.com/ru/post/467039/#1).

**Дизайн**

Самое главное — учесть в код-ревью общий проект (дизайн). Имеют ли смысл взаимодействия разных частей кода? Это изменение относится к вашей кодовой базе или к библиотеке? Хорошо ли CL интегрируется с остальной частью системы? Время ли сейчас добавлять эту функциональность?

**Функциональность**

Делает ли этот CL то, что задумал разработчик? Хорошо ли оно для пользователей этого кода? Под «пользователями» подразумеваются и конечные пользователи (если их затрагивает изменение), и разработчики (которым придётся «использовать» этот код в будущем).  
  
В основном, мы ожидаем, что ещё до коммита разработчики протестируют свой код достаточно хорошо, чтобы он правильно работал. Но как рецензент вы всё равно должны думать о крайних случаях, искать проблемы параллелизма, пытаться думать как пользователь и даже при чтении кода смотреть, что нет очевидных ошибок.  
  
Если хотите, то *можете* проверить работоспособность. Наиболее важно сделать это, если код оказывает влияние на пользователей, например, **изменение UI**. Трудно понять, как некоторые изменения повлияют на пользователей, когда вы просто читаете код. Для таких изменений можете попросить разработчика предоставить демо, если вам слишком сложно углубляться в код и испытать его самостоятельно.  
  
Ещё один момент, когда во время код-ревью особенно важно подумать о функциональности, — это если в CL происходит какое-то **параллельное программирование**, которое теоретически может вызвать взаимоблокировки или условия гонки. Такие проблемы очень трудно обнаружить, просто запустив код; обычно нужно, чтобы кто-то (и разработчик, и рецензент) тщательно продумали их и убедились, что проблемы не вводятся (обратите внимание, что это также хорошая причина не использовать модели параллелизма, где возможны условия гонки или взаимоблокировки, — это может сделать код очень сложным для понимания или код-ревью).

**Сложность**

Является ли CL более сложным, чем должен быть? Проверьте это на каждом уровне: отдельные строки, функции, классы. «Излишняя сложность» обычно означает **невозможность быстрого понимания при чтении**. Это также может означать, что **разработчики, скорее всего, будут вводить ошибки при попытке вызвать или изменить этот код**.  
  
Особый тип сложности — это оверинжиниринг, когда разработчики сделали код более универсальным, чем он должен быть, или добавили функциональность, которая в настоящее время не нужна системе. Рецензентам следует быть особенно бдительными в отношении оверинжиниринга. Поощряйте разработчиков решать проблему, которая точно должна быть решена сейчас, а не проблему, которую, возможно, потребуется решить в будущем. Будущую проблему следует решать тогда, когда она появится, и вы можете увидеть её фактическую форму и требования в физической Вселенной.

**Тесты**

Запросите модульные, интеграционные или сквозные тесты, соответствующие изменению. В общем случае тесты следует добавить в тот же CL, что и производственный код, если CL не обрабатывает [чрезвычайную ситуацию](https://google.github.io/eng-practices/review/emergencies.html).  
  
Убедитесь, что тесты правильны, разумны и полезны. Тесты не проверяют сами себя, и мы редко пишем тесты для наших тестов — человек должен сам убедиться, что тесты валидны.  
  
Действительно ли тесты не проходят на сломанном коде? Если код изменится, не появятся ли ложные срабатывания? Делает ли каждый тест простые и полезные утверждения? Правильно ли тесты разделены между различными методами тестирования?  
  
Помните, что тесты — код, который тоже придётся поддерживать. Не допускайте в них сложности только потому, что это не часть основного двоичного файла.

**Именование**

Разработчик везде выбрал хорошие имена? Хорошее имя достаточно длинное, чтобы полностью передать то, чем является или что делает элемент, не будучи настолько длинным, что становится трудно читать.

**Комментарии**

Написал ли разработчик чёткие комментарии на понятном языке? Действительно ли все комментарии необходимы? Обычно комментарии полезны, когда они объясняют, почему какой-то код существует, и не должны объяснять, что делает этот код. Если код недостаточно ясен, чтобы объяснить себя, то подлежит упрощению. Есть некоторые исключения (например, комментарии с объяснением действия кода часто бывают очень полезны для регулярных выражений и сложных алгоритмов), но в основном комментарии предназначены для информации, которую не может содержать сам код, например, обоснования решения.  
  
Также может быть полезно посмотреть на комментарии в предыдущем коде. Возможно, есть TODO, который сейчас можно удалить, или комментарий, который не советует вводить это изменение, и т. д.  
  
Обратите внимание, что комментарии отличаются от *документации* по классам, модулям или функциям, которая описывает задачу кода, как он должен использоваться и как он себя ведёт в работе.

**Стиль**

У нас в Google есть [руководства по стилю](http://google.github.io/styleguide/) для всех основных языков, и даже для большинства второстепенных. Убедитесь, что CL не противоречит соответствующим руководствам по стилю.  
  
Если хотите улучшить некоторые элементы, которых нет в руководстве по стилю, добавьте к комментарию соответствующее примечание (*Nit:*). Разработчик будет знать, что это ваша личная ремарка, которая не является обязательной к исполнению. Не блокируйте отправку коммита только на основе личных предпочтений.  
  
Автор не должен совмещать значительные изменения стиля с другими изменениями. Это затрудняет просмотр изменений в CL, усложняет слияния, откаты кода и вызывает другие проблемы. Например, если автор хочет переформатировать весь файл, попросите оформить смену стиля в одном CL, а после этого отправить CL с функциональными изменениями.

**Документация**

Если выход CL изменяет что-то в сборке, тестировании, процедурах взаимодействия или выпуска кода, проверьте факт обновления соответствующей документации, в том числе файлов README, страниц g3doc и всех генерируемых справочных документов. Если CL удаляет код или делает его устаревшим, подумайте, следует ли также удалить документацию. Если документация отсутствует, запросите её создание.

**Каждая строка**

Просмотрите *каждую* строку в коде. Хотя какие-нибудь файлы данных, сгенерированный код или большие структуры данных можно просмотреть мельком, но внимательно прочитайте каждый написанный человеком класс, функцию или блок кода, никогда не предполагайте по умолчанию, что там всё в порядке. Очевидно, что какой-то код заслуживает более тщательного изучения, чем другой — вы решаете сами, но вы должны по крайней мере быть уверены, что *понимаете* работу всего кода.  
  
Если вам слишком сложно читать код и это замедляет ревью, то следует сообщить разработчику и подождать, пока он внесёт ясность, прежде чем продолжить работу. В Google мы нанимаем замечательных инженеров-программистов, и вы один из них. Если вы не можете понять код, очень вероятно, что другие разработчики тоже не смогут. Таким образом, вы также помогаете будущим разработчикам понять этот код, когда просите разработчика внести ясность.  
  
Если код понятен, но вы не чувствуете в себе достаточной квалификации для оценки какого-то фрагмента, убедитесь, что в CL есть рецензент, который квалифицирован, особенно для сложных проблем, таких как безопасность, параллелизм, доступность, интернационализация и т. д.

**Контекст**

Часто бывает полезно взглянуть на CL в широком контексте. Обычно инструмент код-ревью показывает только несколько строк возле места изменения. Иногда нужно посмотреть на весь файл, чтобы убедиться, что изменение действительно имеет смысл. Например, вы видите добавление всего четырёх строк, но если посмотреть на весь файл, то эти четыре строки находятся в 50-строчном методе, который теперь действительно придётся разбить на более мелкие.  
  
Также полезно думать о CL в контексте системы в целом. Улучшает ли он качество кода системы или делает её более сложной, менее протестированной и т. д.? **Не принимайте коммит, который снижает качество кода системы.** Большинство систем усложняются в результате суммы многих небольших изменений, поэтому важно предотвратить там даже небольшие сложности.

**Хорошее**

Если видите что-то хорошее в коммите, сообщите разработчику, особенно когда он в лучшем виде решил проблему, изложенную в одном из ваших комментариев. Код-ревью часто просто фокусируются на ошибках, но они также должны поощрять и ценить хорошие практики. С точки зрения менторинга иногда даже более важно сказать разработчику, что именно он сделал правильно, а не где ошибся.

**Резюме**

При выполнении код-ревью следует убедиться, что:

* Код хорошо спроектирован.
* Функциональность хорошая для пользователей кода.
* Любые изменения UI разумны и выглядят хорошо.
* Любое параллельное программирование выполняется безопасно.
* Код не более сложен, чем должен быть.
* Разработчик не реализует то, что *может* понадобиться в будущем с неизвестными перспективами.
* Код обложен соответствующими модульными тестами.
* Тесты хорошо разработаны.
* Разработчик везде использовал чёткие имена.
* Комментарии понятны и полезны, и в основном отвечают на вопрос *почему?*, а не *что?*
* Код надлежащим образом документирован (как правило, в g3doc).
* Код соответствует нашим руководствам по стилю.

Во время ревью обязательно просмотрите **каждую строку** кода, посмотрите на **контекст**, убедитесь, что вы **улучшаете качество кода** и хвалите разработчиков за то **хорошее**, что им удалось сделать.

**Навигация по CL в код-ревью**

**Резюме**

Теперь, когда вы знаете, [что проверять в коде](https://habr.com/ru/post/467039/#2), каков наиболее эффективный способ проведения код-ревью на нескольких файлах?

1. Имеет ли смысл данное изменение? Есть ли у него хорошее [описание](https://google.github.io/eng-practices/review/developer/cl-descriptions.html)?
2. Сначала посмотрите на самую важную часть. Хорошо ли она спроектирована?
3. Посмотрите на остальную часть CL в соответствующей последовательности.

**Шаг первый: окиньте взглядом весь коммит целиком**

Посмотрите на [описание CL](https://google.github.io/eng-practices/review/developer/cl-descriptions.html) и на то, что он делает в целом. Имеет ли это изменение вообще смысл? Если его изначально не следовало писать, пожалуйста, немедленно ответьте с объяснением, почему оно не нужно. Когда отклоняете такое изменение, также неплохо предложить разработчику, что сделать вместо него.  
  
Например, вы можете сказать «Похоже, вы проделали хорошую работу, спасибо! Тем не менее, мы на самом деле собираемся удалять систему FooWidget, поэтому не хотим вносить какие-то новые изменения прямо сейчас. Как насчёт того, чтобы вместо этого вы провели рефакторинг нашего нового класса BarWidget?»  
  
Обратите внимание, что рецензент не только отклонил текущий CL и предоставил альтернативное предложение, но и сделал это *вежливо*. Такая вежливость важна, потому что мы хотим показать, что уважаем друг друга как разработчики, даже когда не согласны друг с другом.  
  
Если вы получаете довольно много нежелательных CL, то следует пересмотреть процесс разработки в вашей команде или опубликованные правила для внешних контрибуторов, чтобы улучшить коммуникацию при написании CL. Лучше сказать людям «нет» раньше, чем они проделают тонну работы, которую придётся выбросить или сильно переписать.

**Шаг второй: изучите основные части CL**

Найдите файл или файлы, которые представляют «основную» часть этого CL. Часто существует один файл с наибольшим количество логических изменений, и это основная часть CL. Сначала посмотрите на эти основные части. Это помогает понять контекст меньших частей CL и, как правило, ускоряет выполнение код-ревью. Если CL слишком велик, спросите разработчика, что посмотреть сначала, или попросите его [разделить CL на части](https://google.github.io/eng-practices/review/developer/small-cls.html).  
  
Если видите серьёзные проблемы в главной части CL, вы должны немедленно отправить эти комментарии, даже если у вас нет времени, чтобы просмотреть остальную часть CL прямо сейчас. Фактически, просмотр остальной части может оказаться пустой тратой времени, потому что если проблемы достаточно значительны, многие другие рассматриваемые фрагменты кода исчезнут и не будут иметь значения.  
  
Есть две основные причины, по которым так важно немедленно отправить эти основные комментарии:

* Разработчики часто отправляют CL, а затем сразу начинают новую работу на его основе, пока ждут результата код-ревью. Если в CL серьёзные проблемы, им придётся переработать и следующий CL. Желательно выявить ошибки как можно раньше, прежде чем они сделали много дополнительной работы поверх проблемного дизайна.
* Серьёзные изменения занимают больше времени, чем небольшие правки. Почти у всех разработчиков есть дедлайны. Чтобы уложиться в них и не снизить качество кода, любой крупный рефакторинг следует начинать как можно скорее.

**Шаг третий: просмотрите остальную часть CL в соответствующей последовательности**

Убедившись в отсутствии серьёзных проблем с дизайном CL в целом, попробуйте выяснить логическую последовательность для просмотра файлов и убедитесь, что ничего не пропустите. Обычно когда вы просмотрели основные файлы, проще всего просто пройти все остальные в том порядке, в котором их представляет инструмент код-ревью. Также иногда полезно сначала прочитать тесты, а потом основной код, потому что тогда у вас будет представление о том, в чём смысл изменения.

**Скорость код-ревью**

**Один рабочий день — максимальное время для ответа** (т. е. это первое дело на следующее утро).  
  
Следование этим рекомендациям означает, что типичный CL должен получить несколько раундов ревью (если это необходимо) в течение одного дня.

**Скорость и отвлечения**

Есть один момент, когда приоритет личной скорости превосходит командный. **Если вы в середине сосредоточенной задачи, такой как написание кода, не отвлекайтесь на код-ревью.** Исследования показали, что у разработчика может занять много времени, чтобы вернуться в плавный поток разработки после отвлечения. Таким образом, отвлечение от кодирования на самом деле обходится команде дороже, чем просьба к другому разработчику немного подождать код-ревью.  
  
Вместо этого дождитесь точки останова в своей работе. Например, после завершения текущей задачи, после обеда, возвращаясь со встречи, возвращаясь с офисной мини-кухни и т. д.

**Быстрые ответы**

Когда мы говорим о скорости код-ревью, нас интересует время *ответа*, а не то, сколько времени требуется на весь процесс до конца. В идеале, весь процесс тоже должен быть быстрым, но **даже важнее, чтобы быстро приходили *отдельные ответы*, чем весь процесс**.  
  
Даже если весь *процесс* код-ревью требует много времени, наличие быстрых ответов от рецензента на протяжении всего процесса значительно облегчает жизнь разработчикам, которые могут испытывать раздражение от «медленных» ответов.  
  
Если вы слишком заняты для полного ревью сразу после получения запроса, то всё равно можете быстро ответить с сообщением о сроках или предложить ревью другим рецензентам, которые могут ответить быстрее, или [предоставить некоторые первоначальные общие комментарии](https://google.github.io/eng-practices/review/reviewer/navigate.html) (примечание: ничто из этого не означает, что вы должны прервать кодирование даже для отправки такого ответа — отправьте ответ во время разумной точки останова в своей работе).

**Код-ревью между часовыми поясами**

При работе между разными часовыми поясами попробуйте ответить автору, пока он ещё в офисе. Если он уже ушёл домой, то обязательно постарайтесь отправить ответ прежде, чем он вернётся в офис на следующий день.

**LGTM с оговорками**

Ради повышения скорости существуют определённые ситуации, в которых рецензент должен дать LGTM/одобрение, даже в случае неотвеченных комментариев к CL. Это делается в случае, если:

* Рецензент уверен, что разработчик надлежащим образом рассмотрит все оставшиеся комментарии.
* Остальные изменения незначительны и *необязательны*.

Рецензент должен указать, какой из этих вариантов он имеет в виду, если это не ясно.  
  
LGTM с оговорками особенно полезны, когда разработчик и рецензент находятся в разных часовых поясах, а в противном случае разработчик будет ждать целый день, чтобы получить «LGTM, одобрено».

**Большие CL**

Если кто-то присылает вам код-ревью настолько большой, что вы не уверены, когда сможете его просмотреть, то типичный ответ должен состоять в том, чтобы попросить разработчика [разделить CL на несколько меньших CL](https://google.github.io/eng-practices/review/developer/small-cls.html). Обычно это возможно и очень полезно для рецензентов, даже если требует дополнительной работы от разработчика.  
  
Если CL нельзя разбить на более мелкие CL и у вас нет времени быстро просмотреть всё это, по крайней мере, напишите несколько комментариев к общему дизайну CL и отправьте их обратно разработчику для улучшения. Одна из ваших целей как рецензента должна заключаться в том, чтобы всегда «разблокировать» разработчика или позволить ему быстро предпринять какие-либо дальнейшие действия, не жертвуя качеством кода.

**Улучшения код-ревью со временем**

Если будете следовать этим рекомендациям и строго подходить к код-ревью, то обнаружите, что весь процесс со временем ускоряется и ускоряется. Разработчики узнают, что требуется для качественного кода, и отправляют вам CL, которые с самого начала отлично подходят, требуя все меньше и меньше времени на просмотр. Рецензенты учатся быстро реагировать и не добавлять ненужные задержки в процесс рецензирования. Но не компрометируйте стандарты обзора кода или качество для воображаемого улучшения скорости — так на самом деле вы не добьётесь общего ускорения в долгосрочной перспективе.

**Чрезвычайные ситуации**

Есть ещё [чрезвычайные ситуации](https://google.github.io/eng-practices/review/emergencies.html), когда CL должны очень быстро пройти через весь процесс код-ревью, и где придётся смягчить принципы качества. Но пожалуйста, ознакомьтесь с [описанием](https://google.github.io/eng-practices/review/emergencies.html#what), какие ситуации квалифицируются как чрезвычайные, а какие нет.