DOI 10.37882/2223-2966.2025.06-2.47

СРАВНИТЕЛЬНЫЙ АНАЛИЗ МОДЕЛЕЙ ОРГАНИЗАЦИИ РАБОТЫ КОМАНДЫ В СИСТЕМАХ УПРАВЛЕНИЯ ВЕРСИЯМИ

COMPARATIVE ANALYSIS OF TEAM ORGANIZATION MODELS IN VERSION CONTROL SYSTEMS

M. Shakhov

Summary. This article is dedicated to a comparative analysis of team organization models in version control systems. The article focuses on the distributed Git system, but the models considered can be adapted to other systems, such as Mercurial or SVN. This topic is relevant because even today it is not exactly known which factors should influence the choice of a particular branching model, as well as how to properly adapt these models to the specific conditions of the team. The author reviews the advantages and disadvantages of the approaches, their structure and user experience. As a result of the study, recommendations for choosing and integrating the most effective model based on the specifics of the project, its infrastructure and the maturity level of the team are formulated. Additionally, attention was paid to the scalability of the models and their interchangeability due to similar development processes. The results of the study can be used to optimize development processes in agile teams and when implementing various DevOps practices. The article will be useful for both junior-developers and experienced specialists in development management.

Keywords: version control system, branching, development management, git, gitflow, trunk-based development.

Шахов Максим Андреевич
Аспирант, Российский государственный университет имени А.Н. Косыгина
maxim.shkhv@yandex.ru

Аннотация. Данная статья посвящена сравнительному анализу моделей организации работы команды в системах управления версиями. Акцент сделан на распределенной системе Git, однако рассмотренные модели могут быть адаптированы и для других систем, таких как Mercurial или SVN. Актуальность данной темы обусловлена тем, что на сегодняшний день недостаточно изучен вопрос того, какие именно факторы должны влиять на выбор той или иной модели ветвления, а также как правильно адаптировать эти модели под конкретные условия работы команды. Автор проводит анализ достоинств и недостатков таких популярных подходов как GitFlow и Trunk-Based Development, их структуры и особенности эксплуатации. В результате исследования сформулированы рекомендации по выбору и внедрению в проект наиболее эффективной модели, основываясь на специфике проекта, имеющейся инфраструктуре, а также уровне зрелости команды. Помимо этого, уделено внимание вопросам масштабируемости моделей и их взаимозаменяемости за счет схожих процессов разработки. Результаты исследования могут быть использованы для оптимизации процессов разработки в agile-командах и при внедрении различных DevOps-практик. Статья будет полезна как для начинающих разработчиков, так и для опытных специалистов по управлению командной разработкой.

Ключевые слова: система управления версиями, ветвление, управление разработкой, git, gitflow, trunk-based development.

Введение

Современная разработка программного обеспечения требует эффективных подходов к управлению версиями и организации работы команд. В условиях высокой динамики изменений и необходимости обеспечения стабильности кодовой базы особую актуальность приобретает выбор оптимальной модели ветвления в системах контроля версий, таких как Git, Mercurial и SVN. На сегодняшний день в Git наибольшее распространение получили две методологии: GitFlow и Trunk-Based Development (TBD), каждая из которых предлагает различные стратегии управления ветками, интеграции изменений и выпуска релизов.

С практической точки зрения, необходимо сформулировать четкие рекомендации для менеджеров и разработчиков, которые помогут им выбрать наиболее подходящую модель ветвления еще на старте, а также поддерживать ее в актуальном состоянии в процессе работы над проектом.

Материалы и методы

Git, как и другие системы контроля версий предназначен для отслеживания изменений в коде проекта и координации работы членов команды. Однако, в отличии от своих аналогов, Git обеспечивает децентрализованное хранение репозитория, что позволяет участникам команды работать автономно, сохраняя полную историю изменений и возможность синхронизации с общим кодом. Ключевые функции Git включают фиксацию (commit) изменений, управление историей версий, слияние (merge) модификаций от разных разработчиков, а также откат к предыдущим состояниям проекта при необходимости. Использование Git в современных IT-проектах является стандартом де-факто, так как оно способствует повышению прозрачности разработки, минимизирует риски потери кода и в целом упрощает совместную разработку [1, с. 304].

Ветвление (branching) является одним из ключевых механизмов Git, позволяющим создавать параллельные,

изолированные линии разработки, называемые ветками. Это позволяет разработчикам работать над различными задачами одновременно, при этом не мешая друг другу. Помимо параллельного внесения изменений, данный механизм позволяет проводить тестирование, создавать релизы, а также находить и исправлять ошибки. Эффективное использование ветвления требует выбора оптимальной модели, которая будет соответствовать специфике проекта и команде разработчиков [8, с. 87].

Наиболее известными моделями организации работы команды являются GitFlow и Trunk-Based Development (TBD). GitFlow, предложенный Винсентом Дриссеном в 2010 году, представляет собой структурированный подход с жестким разделением веток для разработки, тестирования и выпуска.

В своей основе GitFlow предполагает наличие двух основных веток: master (или main) для стабильной версии кода и develop для активной разработки. Помимо этого, в проекте создаются вспомогательные ветки для исправлений ошибок (hotfix) и резизов (release). Основной же функционал проекта разрабатывается в ветках feature/*, которые создаются от develop и сливаются обратно в нее после завершения работы над функционалом [4, с. 3]. Таким образом, GitFlow позволяет команде работать над несколькими функциональными изменениями одновременно. При этом каждая feature-ветка является изолированной и перед слиянием ее код должен быть проверен, а само изменение протестировано как в автоматическом режиме, так и вручную. Такой подход позволяет назначать ответственного разработчика для каждой ветки, в то время как ревью ветки может быть проведено любым другим членом команды [5].

Однако, несмотря на популярность, GitFlow имеет два главных недостатка:

1. В модели никак не обговаривается то, какой объем задач можно выполнить в рамках одной ветки. Это может привести к тому, что разработчики будут создавать слишком большие ветки, которые другим членам команды будет сложно вовремя проверять и тестировать, поскольку у них тоже могут быть большие задачи в других ветках.
2. Из первого пункта вытекает то, что в проекте будут реже проводиться интеграции. Это может привести к тому, что в ветках будет появляться большое количество конфликтов, которые придется решать в момент слияния, что может еще больше забирать время у других членов команды.

Помимо GitFlow существуют его производные, такие как GitHub Flow и GitLab Flow. Эти модели предлагают более гибкие подходы к ветвлению и интеграции изменений, что позволяет командам адаптировать их под свои нужды. Например, GitHub Flow предполагает использование только одной основной ветки (master или develop). При этом модель хорошо интегрируется с CI/CD-практиками, которые предоставляет сам GitHub, так как все feature-фетки сливаются через Pull Request, который позволяет проводить непрерывную интеграцию [9, с. 192]. Помимо этого, в отличии от GitFlow, в GitHub Flow после слияния ветки, изменения сразу попадают в продакшн.

Другой распространенной моделью, которую часто ставят в противовес GitFlow, является Trunk-Based Development (TBD). Эта модель предполагает, что разработка ведется в одной основной ветке trunk (название не имеет значения, поскольку она одна, поэтому иногда может называться master/main или develop), а все изменения интегрируются в нее как можно быстрее [10].

Основными особенностями TBD являются:

1. Короткоживущие feature-ветки (обычно не больше 1–2 дней), позволяющие разработчикам быстро интегрировать изменения в основную ветку;
2. Частые коммиты (несколько раз в день) в основную ветку. Вытекает из первого пункта;
3. Обязательная CI/CD-интеграция с автоматизированным тестированием каждого коммита. Без этого быстрая интеграция изменений в основную ветку могла бы провоцировать нестабильность кодовой базы;
4. Feature-флаги для управления незавершенным функционалом;

Короткоживущие feature-ветки и как следствие быстрые интеграции позволяют команде поддерживать актуальную версию кода и минимизировать количество конфликтов при слиянии. Для этого TBD также предполагает использование feature-флагов (сокр. FF), которые позволяют включать или отключать функционал по мере необходимости, что упрощает тестирование и развертывание новых функций. При помощи feature-флагов можно скрывать незавершенный функционал от продакшена, что позволяет разделять одну большую задачу на несколько маленьких, которые можно будет интегрировать в основную ветку по мере готовности. Именно такой подход дает возможность производить изменения в короткоживущих ветках [2, с. 170].

Очевидным плюсом короткоживущих веток является то, что их значительно легче проверять и тестировать. Из-за меньшего количество изменений в ветке ревью кода могут проводить все члены команды, а не только самые опытные разработчики. Это позволяет избежать ситуации, когда один разработчик становится узким местом в процессе разработки, так как он является единственным, кто может проверить и слить ветку в trunk. Такой подход также повышает вовлеченность всех членов команды в работу над кодовой базой и способствует

обмену знаниями между разработчиками. В этом плане TBD схож с методологией Extreme Programming (XP), которая также акцентирует внимание на совместной работе команды и частых интеграциях.

Для обеспечения темпа разработки в рамках короткоживущих веток, от команды требуется навык разбития одной целостной задачи на несколько мелких. В рамках TBD такая техника называется branch by abstraction (англ. ветвление по абстракции). Ее суть заключается в том, что разработчик или команда создает абстракцию для нового функционала, а затем постепенно реализует его, не нарушая работу существующего кода [3]. В качестве примера можно рассмотреть задачу по замене старого модуля аутентификации *AuthAdapter* на новый. Данную задачу можно разбить на несколько этапов:

1. Создание интерфейса *AuthProvider* и его имплементация для нового и старого модуля;
2. Внедрение механизма выбора реализации (старой или новой) с помощью feature-флагов;
3. Через инструменты для A/B-тестирования часть трафика перенаправляется на новый модуль. У нового модуля анализируются потенциальные ошибки и скорость работы;
4. Если все прошло успешно, то старый модуль можно удалить, а новый оставить как основной.

Данный метод минимизирует риски, в особенности из-за того, что новый функционал можно легко отключить через feature-флаги, если что-то пойдет не так [7, с. 532]. Однако, у такого подхода в рамках TBD есть два основных недостатка:

1. Сложность организации задач по branch by abstraction менее опытными разработчиками. Для того, чтобы эффективно использовать данный метод, разработчики должны хорошо понимать архитектуру приложения и иметь высоко развитое абстрактное мышление.
2. Отсутствие четких рекомендаций по удалению feature-флагов. В процессе работы над проектом может накопиться большое количество feature-флагов, которые могут негативно сказаться на производительности приложения и усложнять кодовую базу. Это особенно актуально при разработке frontend-приложений, где отключенные при помощи флагов части кода все равно попадают в сборку проекта.

Результаты и обсуждения

Проводя сравнение двух основных моделей ветвления, можно сделать вывод о том, что для выбора оптимальной модели необходимо учитывать особенности команды, проекта и его архитектуры. В первую очередь следует ориентироваться на зрелость команды и ее координацию. Если команда состоит из опытных разработчиков, которые ориентируются в архитектуре приложения, а team lead способен использовать абстракции для разбития задач на подзадачи, то TBD будет оптимальным выбором. Это позволит команде поддерживать высокую скорость разработки и минимизировать количество конфликтов при слиянии. Однако, если команда состоит из новичков или разработчиков с низким уровнем координации, то использование TBD может привести к проблемам с интеграцией изменений и значительному увеличению времени на постановку задач.

Для команд с низкой координацией как правило больше подходит GitFlow, поскольку он описывает лишь сами правила ветвления и не затрагивает процессы постановки задач, а также разработки напрямую. GitFlow имеет более низкий порог вхождения, поскольку управлением ветками, интеграцией и ревью занимается team lead или более опытные senior-разработчики. Это позволяет новичкам сосредоточиться на написании кода, не отвлекаясь на организацию самих процессов разработки. Однако, стоит помнить и о том, что таким образом большая часть ответственности за качество кода ложится на плечи очень узкого круга разработчиков.

Другим критерием выбора модели является зрелость DevOps процессов в проекте (например, наличие качественного CI/CD). Если команда уже использует настроенную CI/CD инфраструктуру, то интеграция TBD в процесс разработки не составит труда, поскольку развитое автоматическое тестирование позволит с наименьшими усилиями проводить регулярные слияния.

При внедрении в проект TBD необходимо настроить поддержку feature-флагов, при помощи которых можно будет отключать незавершенные изменения и проводить A/B тестирование. Важно помнить, что feature-флаги не должны оставаться в кодовой базе на постоянной основе. Они должны храниться в коде только до тех пор, пока функционал не будет полностью завершен и протестирован. После этого флаги должны быть удалены. Стоит отметить, что модель TBD не описывает как именно должно проходить удаление устаревших флагов и решение этой задачи падает на плечи team lead или менеджмента проекта. Например, можно удалять ненужные флаги по завершению релиза, или раз в определенное время (например, раз в месяц). Первый вариант более предпочтителен для проектов, ориентированных на frontend, где при сборке проекта отключенные при помощи флагов части кода все равно попадают в финальную сборку (бандл) приложения, что может негативно сказаться на скорости загрузки и рендеринга страниц, а как следствие и на скоринге производительности (например, Google Lighthouse), который отвечает за SEO-продвижение.

Помимо решения вопроса о feature-флагах, необходимо провести инструктаж команды. Далеко не все даже

опытные разработчики хорошо знакомы с моделью TBS, поэтому следует разъяснить основные принципы работы в ее рамках. Важно донести команде, что основной задачей этой модели являются частые интеграции, из которых и вытекают все ее преимущества.

При выборе GitFlow в качестве модели ветвления необходимо в первую очередь настроить инфраструктуру в виде всех необходимых веток (master, develop, release, hotfix и т.д.). Время интеграции как правило не регламентируется, но стоит обозначить какие-то рамки, чтобы разработчики не создавали чрезмерно большие ветки.

Стоит помнить, что GitFlow также имеет раннее рассмотренные в статье производные (GitHub Flow и GitLab Flow). При этом GitHub Flow является наиболее простой разновидностью GitFlow, которая не требует не только специфичных навыков у членов команды, но и сложного контроля ветвления со стороны team lead. Это делает его оптимальным выбором для небольших команд, где управлением интеграциями занимается только один человек. При этом нужно отметить, что эта модель требует наличия CI/CD-инфраструктуры, которая будет обеспечивать более частые релизы чем в рамках обычного GitFlow. Таким образом, важной особенностью модели является то, что она наиболее похожа на TBD, поэтому ее возможно использовать в команде с недостаточным опытом, при этом не теряя возможность в будущем легко перейти на TBD.

Данное исследование оставляет открытым вопрос о том, как именно необходимо проводить инструктирование команды при внедрении TBD. Ответ на этот вопрос может потребовать дополнительных исследований и практических экспериментов в рамках реальных проектов. Стоит отметить, что подобные инструкции могут отличаться как в зависимости от уровня зрелости самой команды, так и от уровня профессионализма и роли конкретного разработчика.

Заключение

Модели организации ветвления в системах управления версиями представляют из себя мощный инструмент организации командной разработки. Только правильный подход в выборе модели ветвления позволит команде избежать проблем в будущем и извлекать максимум из своего потенциала. В данной статье были рассмотрены основные модели ветвления, их плюс и минусы, а также сформулированы четкие рекомендации по выбору наиболее подходящей модели в зависимости от уровня команды и инфраструктуры проекта. Важно помнить, что выбор модели ветвления не является перманентным и может меняться в будущем по мере развития продукта.

ЛИТЕРАТУРА

1. Фишерман Л.В. Git. Практическое руководство. Управление и контроль версий в разработке программного обеспечения // Наука и техника, 2022.
2. Capture the Feature Flag: Detecting Feature Flags in Open-Source / Meinicke J., Hoyos J., Hoyos V., Kastner C. // Proceedings of the 17th International Conference on Mining Software Repositories. 2020. pp. 169–173.
3. Fowler M. Branch by Abstraction [Электронный ресурс] // URL: https://martinfowler.com/bliki/BranchByAbstraction.html (дата обращения: 02.03.2025)
4. GitFlow: flow revision management for software-defined networks / Dwaraki A., Seetharaman S., Natarajan S., Wolf T. // In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research. 2015. №6, pp. 1–6.
5. Gitflow workflow [Электронный ресурс] // Atlassian. URL: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (дата обращения: 02.03.2025).
6. Humble J., & Farley D. Continuous Delivery. Addison-Wesley Professional, 2010.
7. Prutchi E., Heleno L. How the adoption of feature toggles correlates with branch merges and defects in open-source projects? // Software: Practice and Experience. 2021. № 52. pp. 506–536.
8. Shakikhanli U., Bilicki V. Optimizing branching strategies in mono and multi-repository environments: a comprehensive analysis // Computer Assisted Methods in Engineering and Science. 2024. № 31. pp. 81–111
9. The github development workflow automation ecosystems / Wessel M., Mens T., Decan A., Mazrae N., // In Software Ecosystems: Tooling and Analytics. 2023. pp. 183–214.
10. Trunk Based Development [Электронный ресурс] // URL: https://trunkbaseddevelopment.com/ (дата обращения: 02.03.2025).

Журнал «Современная наука: актуальные проблемы теории и практики»