Управление версиями для нескольких Agile команд

Автор: Henrik Kniberg

Оригинал статьи: http://www.infoq.com/articles/agile-version-control

Благодарности от сайта agilerussia.ru

Большое спасибо Андрею Бибичеву и Антону Каткову за ревью перевода и очень полезные замечания!

Если у нас есть несколько agile команд, работающих с одной и той же кодовой базой, то как минимизировать риск помешать друг другу? Как удостовериться, что в конце итерации получена чистая, готовая к поставке (releasable) версия? В статье описан пример того, как управлять версиями в agile окружении с несколькими командами. Это схема, к которой мы перешли в компании, описанной в "Scrum and XP from the Trenches".

[От переводчиков:

Перевод книги "Scrum and XP from the Trenches": http://scrum.org.ua/wp-content/uploads/2008/12/scrum_xp-from-the-trenches-rus-final.pdf]

Эта статья не предназначена экспертам по управлению версиями, такие эксперты не найдут в ней ничего нового для себя. Эта статья предназначена остальным, тем из нас, кто просто хочет узнать простые и успешные способы взаимодействия. Она может быть интересна всем, кто вовлечен в agile-разработку, независимо от роли, ведь ветвление и слияние касается всех, а не только ответственного за конфигурации.

В конце есть ссылка на pdf-версию статьи.

Оглавление

Введение

В статье описывается управление версиями в agile окружении с несколькими командами. Я делаю допущение, что вы знакомы с основными элементами Scrum, XP и использованием доски задач (taskboard). Я не изобрел эту схему, она базируется на "mainline модели (модели главной ветви)" или "шаблоне чистого trunk-а (ствола)". Дополнительная информация - в секции Ссылки.

Я написал эту статью, потому что продолжаю сталкиваться с командами, которым действительно нужно что-то в этом роде. Большинству команд модель нравится, как только они начинают понимать ее. Кроме того, это схема, на которую мы перешли в компании описанной в "Scrum and XP from the Trenches". Она действительно помогла нам разрабатывать и выпускать ПО более гибко. Описав модель в легко читаемой форме, мне не придется объяснять ее так часто на доске :o)

Обратите внимание, что это только один из многих шаблонов, а не "серебряная пуля". Если вы решите использовать этот шаблон, вам, возможно, надо будет подгонять его под себя.

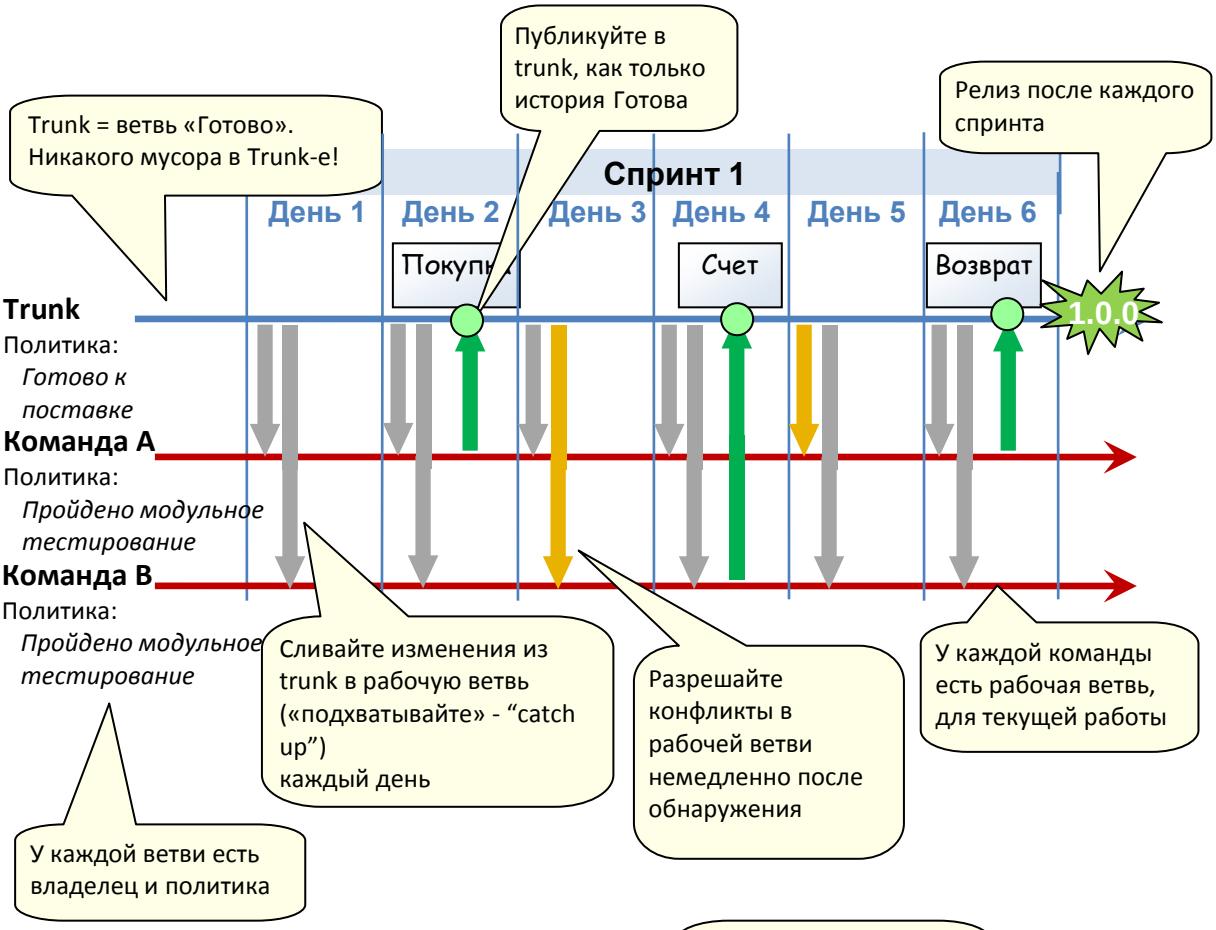
Цели

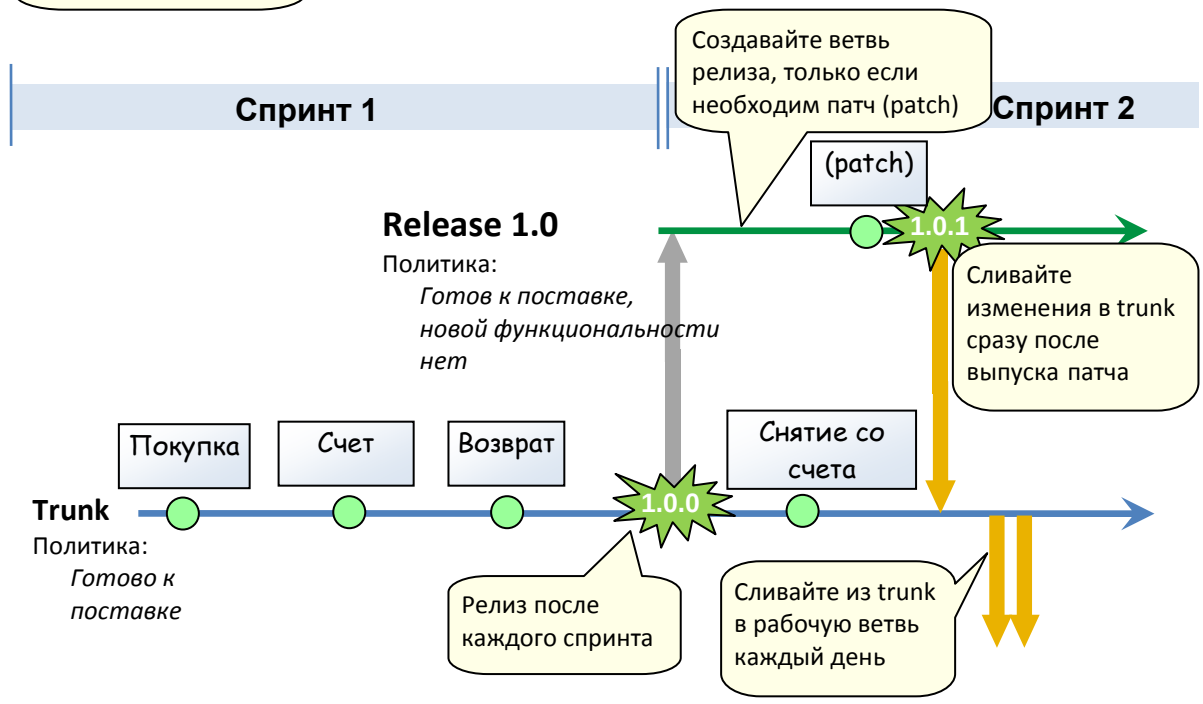
В случае нескольких agile команд модель управления версиями должна удовлетворять следующим требованиям:

- **Быстрое информирование об ошибках**
  - Конфликты кода и проблемы интеграции должны обнаруживаться как можно раньше.
  - Лучше исправлять небольшие ошибки часто, чем большие проблемы редко.
- **Постоянная готовность к поставке**
  - Даже после действительного плохого спринта (sprint) должно быть хоть что-то, что можно выпустить.
- **Простота**
  - Все члены команды будут ежедневно использовать эту схему, поэтому правила должны быть простыми и понятными.

Общая картинка на одну страницу (чтобы повесить на стену)

Если эта картинка сбивает вас с толку, не унывайте, просто читайте дальше.
Если эта картинка очевидна для вас, то для вас чтение этой статьи излишне.





Эту картинку также можно [скачать здесь, в формате pdf](#)

Шаблон управления версиями

Владелец ветви (branch) и политика (policy)

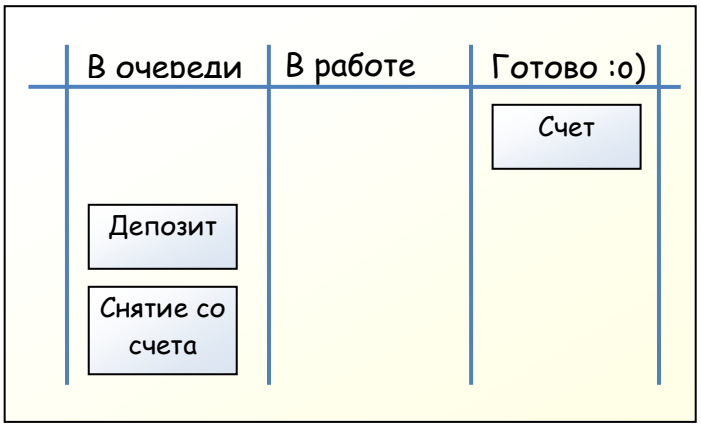
Вот простое правило.

***Правило**: Каждая ветвь (даже основной ствол, т.е. trunk) имеет владельца и политику.*

Иначе будет путаница. Политика описывает правила: что разрешено заливать (check in) в ветвь. Владелец - это человек, который отвечает за определение политики и за следование этой политике.

Концепция "готово" ("done")

Когда можно считать, что пользовательская история (user story) "готова" ("done")? Точнее, когда ваша команда перемещает пользовательскую историю в колонку "Готово" на доске задач, что это действительно означает?



Я делаю следующее допущение.

***Допущение**: Готово - по определению значит готово к поставке.*

Поэтому, когда член команды говорит, что пользовательская история Готова и перемещает ее карточку в колонку Готово, заказчик может вбежать в комнату и сказать "Отлично! Давайте выпускать прямо сейчас!". И никто из членов команды не скажет: "Нет, подождите."

Вы можете использовать то определение Готово, которое вам нравится. Но помните: если это определение означает что-то меньшее, чем готовность к поставке, то вы должны учесть, что не включено в "Готово". И кто это будет делать. Когда? И что будет, если что-то пойдет не так после "Готово"?

Ветвь “Готово” (The Done branch)

Когда пользовательская история готова, ее нужно где-то разместить. По моему определению "Готово" (т.е. "готово к поставке") это означает, что должна быть какая-то ветвь в системе, которую можно поставить, чтобы пользовательская история попала в систему в промышленной эксплуатации (production). Это и есть ветвь “Готово”.

Любая ветвь может быть ветвью “Готово”. Я собираюсь использовать trunk как ветвь “Готово”. Иногда это называют "mainline".

***Допущение**: Trunk - это ветвь “Готово”.*

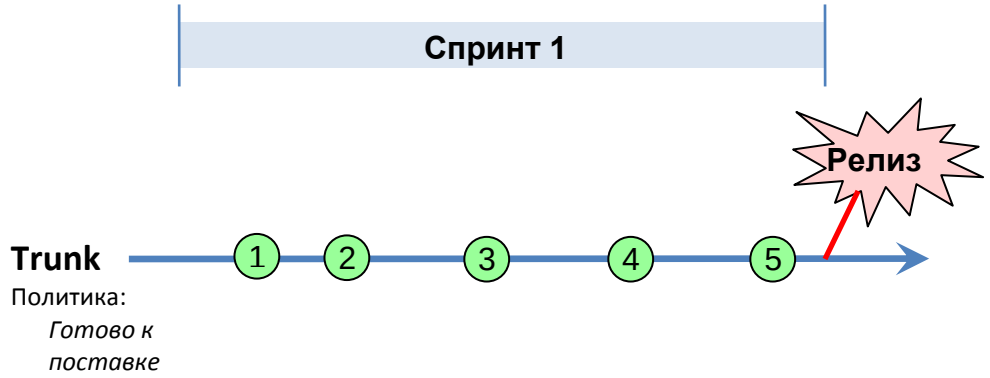
Политика trunk-а:

- Можем поставлять в любое время

- Хотим поставлять сразу по первому требованию

"Можем поставлять в любое время" - означает именно это. В любой момент, когда Владелец Продукта (Product Owner) решит, что мы должны сделать новый релиз с вершины trunk-а.

Вот пример.



Синяя линия представляет trunk. Каждый зеленый кружочек означает одну заливку кода. Т.е. во время спринта заливали код 5 раз. Мы можем сделать поставку из trunk-а в любой момент, хотя обычно мы делаем это в конце каждого спринта. Если полкоманды сляжет ближе к концу спринта и у них не будет времени закончить работу над пользовательской историей #5, мы все равно сможем сделать поставку. И конечно мы можем решить *не* делать поставку и, вместо этого, подождать окончания следующего спринта, чтобы была готова пользовательская история #5.

"Хотим поставлять сразу по первому требованию" - означает, что мы должны залить код для пользовательской истории только если мы *хотим*, чтобы она попала в систему в промышленной эксплуатации (или, по крайней мере, если мы *не против*, чтобы она попала в систему в промышленной эксплуатации). Если пользовательская история #3 с картинки является чем-то таким, что я не готов поставлять, то я, по сути дела, испортил ветвь. Поскольку ветвь больше не готова к поставке, я нарушил политику ветви.

***Правило**: Не совмещайте несколько циклов разработки в одной ветви*

Когда создавать дополнительные ветви?

Как можно реже. Вот хорошее эмпирическое правило.

***Рекомендация**: Создавайте новую ветвь только тогда, когда вы хотите залить код в систему контроля версий , и не существует ветви, которую можно для этого использовать, не нарушив ее политику.*

Рабочие ветви

Хорошо, допустим, у нас есть чистый trunk, готовый к поставке в любой момент.

Погодите секунду. Готовый к поставке - значит, что *интеграция протестирована*. И это значит, что нам нужно *запустить интеграционные тесты*. Т.е. нам нужно запустить интеграционные тесты *до* заливки кода в trunk.

Куда же мне залить код, который, как я *считаю* готов, но требует проверки перед заливкой в trunk? Конечно, я мог бы протестировать его локально, а затем сделать заливку кода в trunk. Но это не очень удачный вариант, я думаю все наталкивались на проблему: "странно, а *у меня* на компьютере это работает нормально!".

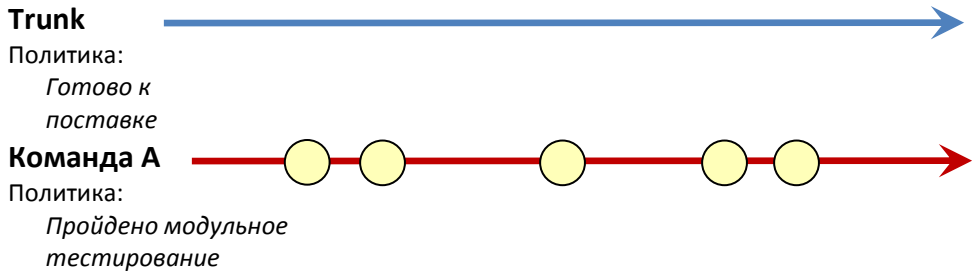
Другая проблема это: "На сегодня я завершил кодирование и иду домой. Куда мне залить свой код? Он пока не протестирован, поэтому я не могу добавить его в trunk. Я хочу внести его так, чтобы другие члены команды смогли продолжить работу над ним." (agile команда = коллективное владение кодом, не так ли?)

Вот что получается. У нас есть код, который мы хотим залить, и нет места, куда мы можем его залить не нарушив политику ветви. Это законная причина для создания новой ветви.

Назовем это *рабочая ветвь* (work branch), совместно используемая всеми членами команды.Некоторые называют это *ветвь разработки* (development branch).

Политика рабочей ветви для команды А:

- Код компилируется и собирается, все модульные (unit) тесты проходят.



Отлично, теперь у нас есть две ветви! Одна стабильная ветвь (trunk) и одна немного менее стабильная (ветвь команды). Ветвь команды красного цвета, чтобы подчеркнуть, что она менее стабильная, т.е. код в этой ветке прошел модульные тесты, но интеграция может быть не протестирована, и код может быть не достаточно стабильным для релиза. ОК, теперь у команды есть место, чтобы заливать код для еще неоконченной работы (work in progress).

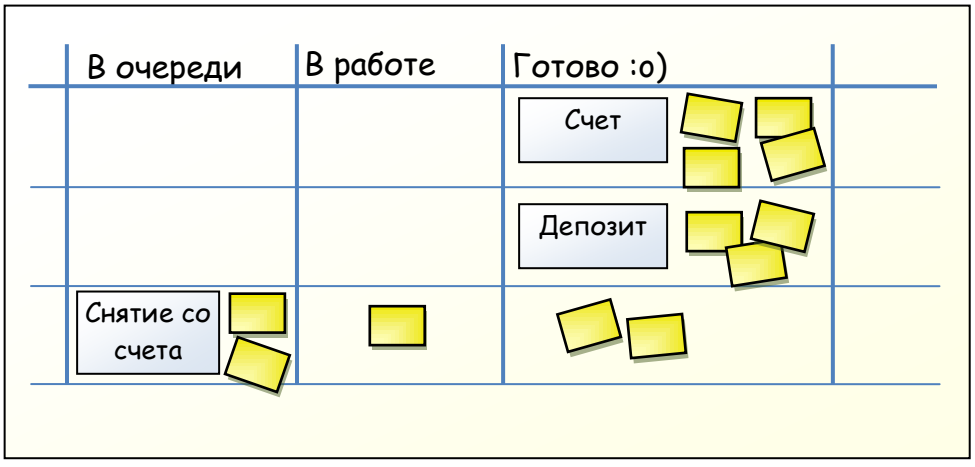
Да.. Как же синхронизировать эти ветки? Читайте далее.

Публикация из рабочей ветви в trunk

В какой-то момент (надеемся) пользовательская история будет Готова. Точнее, в какой-то момент рабочая ветвь будет *готова к поставке (releasable)*. В этот момент мы можем (и должны) перенести ее в trunk, т.е. взять весь новый код из рабочей ветви и скопировать его в trunk. После этого trunk и рабочая ветви станут идентичными.

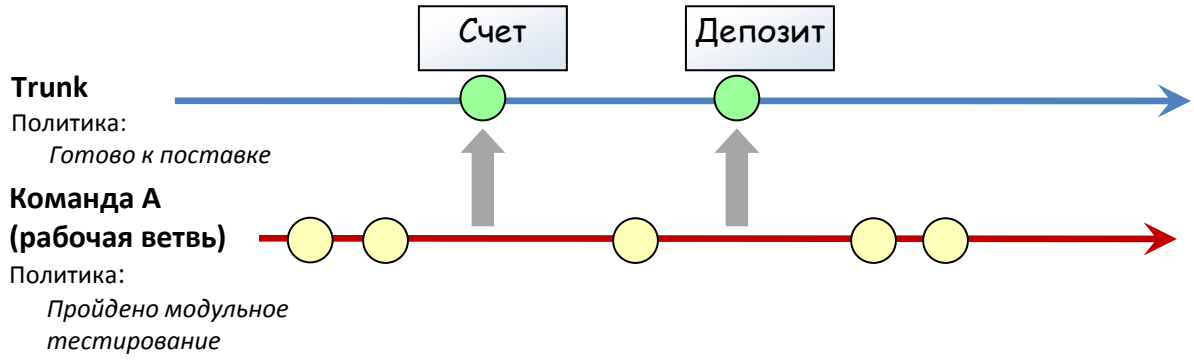
Мы называем это "публикацией", потому что мы сделали некоторую работу и теперь готовы "опубликовать" это обратно в trunk для выпуска. Просто удобная метафора.

Вот пример. Допустим, мы реализовали две пользовательские истории: Счет (Register) и Депозит (Deposit). Они Готовы, т.е. прошли unit тесты, интеграционнные тесты и готовы к поставке. Мы уже начали работать над пользовательской историей Снятие со счета (Withdraw), но она еще не Готова. Доска задач выглядит примерно так:



Каждый желтый листик на доске означает задачу, т.е. один кусочек работы, который нужно сделать, чтобы завершить пользовательскую историю. Например, "изменить класс X", "обновить скрипт для сборки", и т.д. Каждая задача обычно представляет один человеко-день работы, каждая пользовательская история обычно представляет примерно 3 - 8 человеко-дня работы.

История ветки будет выглядеть примерно так:



Итак, сначала наша команда реализует историю Счет. Заливаем код в рабочую ветку, запускаем интеграционные тесты, исправляем ошибки, опять заливаем код, опять запускаем тесты, работает! Пользовательская история Счет готова! Публикуем ее в trunk.

Затем реализуем Депозит. Нужна была только одна заливка кода. Интеграционные тесты пройдены, поэтому мы опять публикуем в trunk.

Сейчас команда в процессе реализации истории Снятие со счета. Они уже два раза залили код, но еще не завершили реализацию.

Заметьте, что публикация в trunk не означает, что мы копируем код для одной определенной пользовательской истории в trunk. Это означает, что мы копируем все из рабочей в trunk, т.е. делаем полную синхронизацию.

Поэтому возникают два интересных вопроса:

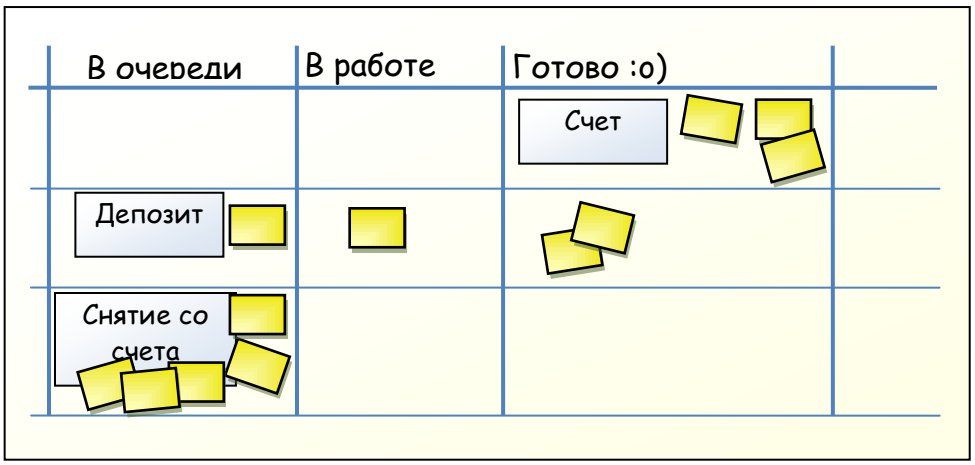
1. Что если наша команда параллельно реализует несколько пользовательских историй?
2. Что если другие команды тоже делают публикацию в trunk?

Давайте разберем эти вопросы один за другим.

Что если наша команда параллельно реализует несколько пользовательских историй?

Если команда реализует одну пользовательскую историю за раз, публикация в trunk тривиальна. Как только история реализована и протестирована в рабочей ветви, мы копируем все из рабочей ветви в trunk. Готово.

Но постойте. Что если команда одновременно реализует несколько пользовательских историй Что если история Счет готова, а Депозит еще в работе?

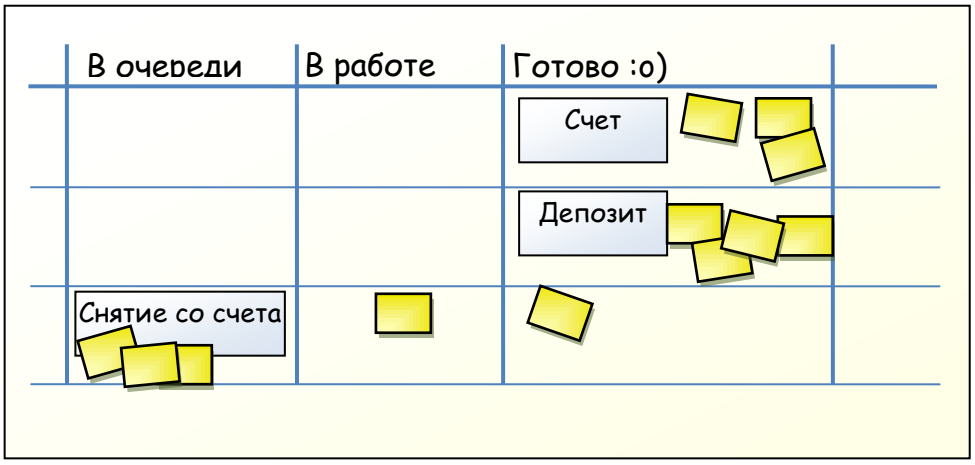


Если мы проведем синхронизацию с trunk-ом в этот момент, мы перенесем в trunk *частично законченную* пользовательскую истрию Депозит, которая не готова к поставке. <Звук сирены> Это нарушение политики ветви! </Звук сирены>

Конечно, мы можем ждать готовности пользовательской истории Депозит.

(..ждем..)

ОК, теперь история Депозит готова! Отлично! Подожните секунду... теперь кто-то начал работать над историей Снятие со счета! Ой! Та же проблема!



Если не пройдет один из тестов на Депозит, то будет сложно понять, произошло ли это из-за кода пользовательской истории Депозит или из-за частично завершенной Снятие со счета, которая была добавлена в ту же ветвь.

Ожидание не помогает. Мы катим большой снежный ком, оставляя все на большой "взрывной" релиз в какой-то гипотетической точке в будущем, когда все пользовательские истории будут готовы (если это вообще когда-нибудь будет).

Это очень распространенная проблема. Так что же нам делать?

Вот некоторые стратегии:

- Не разрабатывать параллельно. Постараться сфокусировать работу всей команды над одной пользовательской историей.
- Если кто-то собирается начать работать над историей Депозит до окончания работ над историей Счет, он не должен заливать код Депозит, пока код Счет не будет готов. Или можно залить код Депозит в отдельную временную ветвь, если вам нравится жонглировать ветвями.
- Если кто-то собирается начать работу над пользовательской историей Депозит до окончания работ над Счет, начните с безопасных и незаметных элементов, с изменений в коде,

которые не повлияют на готовность ветви к поставке. Например, если для истории Депозит нужен новый код и некоторые изменения уже существующего кода, реализуйте новый код (новые методы, новые классы, новые тесты и т.п.), а существующий код не меняйте. Если для истории Депозит нужны новые элементы GUI, сделайте их временно невидимыми. Когда история Счет готова и перенесена в trunk, можно начать реализацию оставшихся кусков пользовательской истории Депозит.

Вот подходящий набор правил:

- Тот, кто работает над самой приоритетной пользовательской историей - Король (King).
- Все остальные в команде - Слуги (Servant).
- Вы хотите быть Королем. Постарайтесь найти способ принять участие в работе над самой приоритетной пользовательской историей.
- Когда Королю нужна помощь, Слуги немедленно предлагают ему свои услуги.
- Слуга не может мешать Королю.
- Слуга не может заливать в рабочую ветвь код, не готовый к поставке. Король может заливать в репозиторий все, что он хочет (конечно, при этом он не должен нарушать политику ветви).
- Как только готова самая приоритетная пользовательская история, Королем становится тот, кто реализует следующую по приоритетности историю.

Вы даже можете достать для этой цели несколько корон :o)

Вообще говоря, многие команды переоценивают преимущества одновременной работы над несколькими пользовательскими историями. Это дает ощущение скорости, но это только иллюзия, поскольку ближе к концу отодвигается реализация рискованных и больших по времени элементов: слияние, интеграция и тестирование.

Поэтому Scrum команды должны быть маленькими (< 9 человек), благодаря этому достигается тесное взаимодействие и команда концентрирует свои усилия. Если каждый занят своей пользовательской историей, то будет не так уж много взаимодействия. Конечно, у вас могут быть люди, которые смотрят вперед, готовят следующую историю и делают кое-что из реализации. Но в любой текущий момент основной объем командных усилий должен быть сосредоточен над самой приоритетной истории.

Другое дело, если есть несколько команд. Несколько команд создаются, когда вы хотите реализовывать несколько пользовательских историй в параллель. Через минуту перейдем к рассмотрению этой ситуации. Но сначала я хочу поговорить про регрессионное тестирование и разошедшийся код в команде.

“Готово” включает в себя регрессионное тестирование!

Когда история Готова мы перемещаем ее в колонку Готово и копируем из рабочей ветви в trunk. Trunk всегда должен быть готов к поставке. Это важное допущение.

***Правило**: Тот, кто вносит изменения в trunk, отвечает за то, что он остается готовым к поставке, включая всю предыдущую функциональность!*

Фактически это правило означает, что тестирование истории А также включает прогон всех связанных регрессионных тестов для уже реализованных до этого историй. Не достаточно того, что работает история А, если при этом сломан код уже реализованных пользовательских историй.

Подождите. Не является ли это безумием? Запускать все регрессионные тесты каждый раз, когда готова очередная story?

Хорошо, во-первых я не говорил, что *все* регрессионные тесты. Я сказал все *связанные* регрессионные тесты. Помните, что у нас есть чистый и готовый к поставке trunk, как базис, и мы просто добавляем одну пользовательскую историю! Это небольшое инкрементальное изменение. Если наши регрессионные тесты автоматизированы, мы можем запустить их все. Но если тесты нужно выполнять вручную, нам придется быть избирательными.

Это все сводится к поиску компромисса между стоимостью и риском. Для каждого регрессионного теста, который выполняется вручную, мы должны сравнить стоимость тестирования (т.е. сколько нужно сделать работы для выполнения теста) и вероятность того, что будут найдены какие-то важные дефекты. И конечно взвесить стоимость автоматизации этого теста :o)

Разошедшийся код (конфликты слияния)

Допустим, я пишу код, который вызывает класс Widget. И я не знаю, что в ходе рефакторинга член моей команды Джим удалил класс Widget час назад, в связи с рефакторингом. Таким образом, у нас есть *разошедшийся код*. Я хочу узнать об этом *как можно раньше*, чтобы не тратить время на написание кода, вызывающего класс Widget.

***Правило**: Постоянно синхронизируйте ваш код с рабочей ветвью (= так часто как это возможно)*

Имеется в виду синхронизация в обоих направлениях. Заберите последние изменения из рабочей ветки, а затем залейте ваш код. Первую часть можно назвать "подхватывание" ("catching up") (= я хочу узнать, что добавили в кодовую базу другие). Вторую часть можно назвать "публикацией" (= я хочу, чтобы сделанные мной изменения были доступны всей команде).

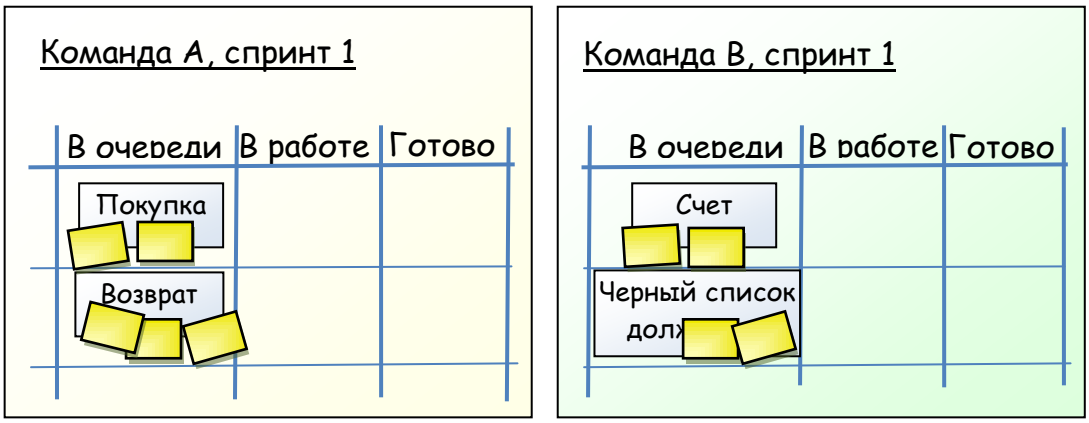
Хорошо бы делать это раз в час или около того. В сущности, каждый раз, когда вы переходите от одной задачи к другой. Это касается не только: "Я хочу как можно раньше узнать, если кто-то пишет код, конфликтующий с моим". Это также: " Я хочу, чтобы другие как можно раньше узнали, если я пишу код, конфликтующий с их кодом". Помните о том, чтобы не нарушать политику рабочей ветви (модульные тесты пройдены, и все остальное).

Это правило может показаться очевидным, но потерпите. Я хочу сделать это абсолютно понятным, поскольку мы будем использовать это мировоззрение далее, разбирая работу с несколькими командами.

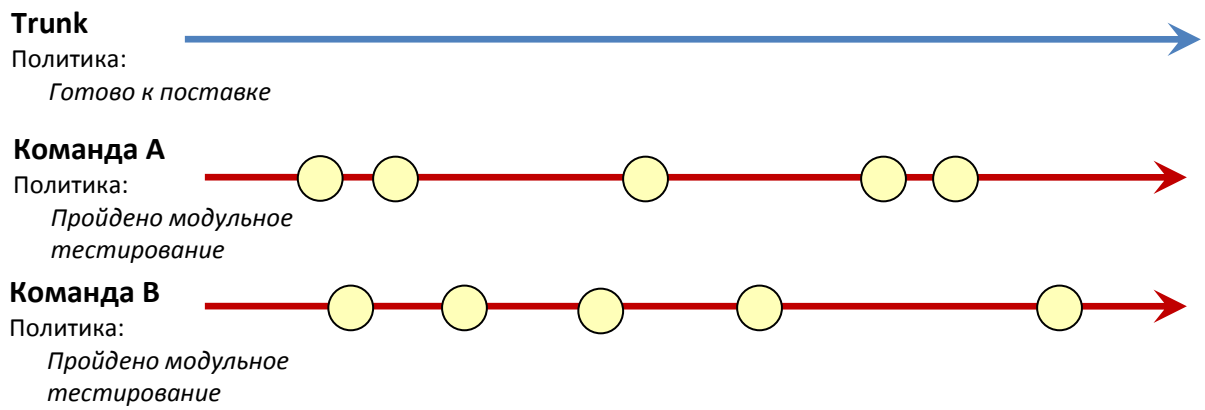
Несколько команд - что если другие команды тоже делают публикацию в trunk?

Допустим, у нас есть команда A и команда B. Это кроссфункциональные команды, которые работают над системой бронирования авиабилетов. Команда A занимается процессом бронирования, команда B занимается бэк-офисом.

Допустим, они собираются сейчас начать спринт, по две пользовательские истории на команду (обычно в спринте больше двух пользовательских историй).



У каждой команды своя рабочая ветвь, поскольку каждой нужно где-то тестировать код перед публикацией в trunk.

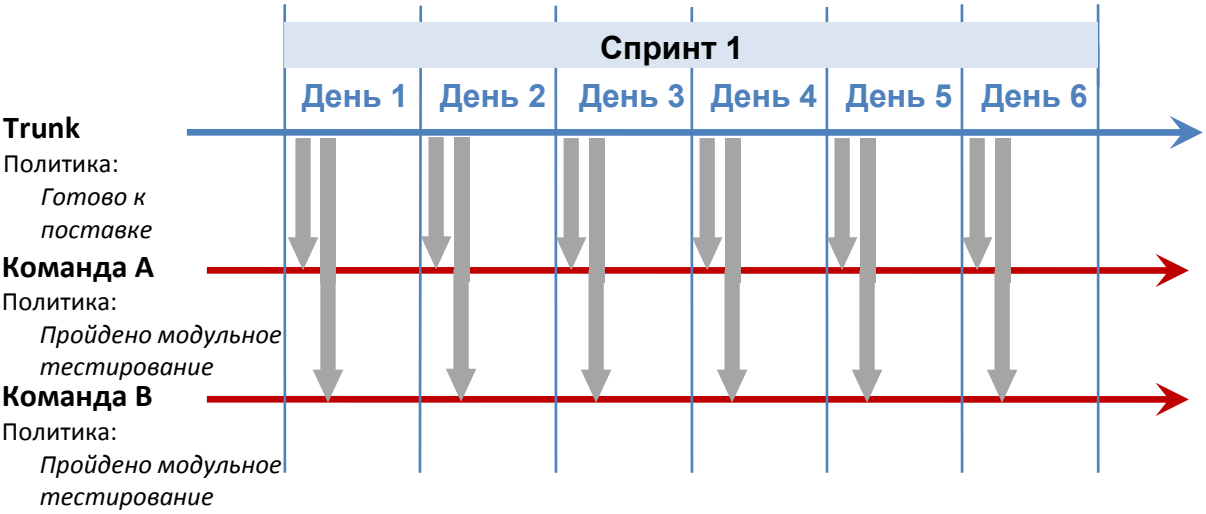


У нас теперь есть интересная проблема. Допустим, я из команды A и у нас есть рабочая ветвь. В trunk-е могли произойти изменения, которые не были изначально сделаны в моей рабочей ветви! Почему? Потому что есть другая команда, и они делают публикацию в trunk, когда заканчивают работу над историей!

Таким образом, в любой момент в trunk-е может появиться код, о котором я не знаю. И этот код может конфликтовать с моим кодом! Возможно кто-то в команде B переименовал класс Widget, который я вызываю из своего кода и .. ух.. подождите. Разве мы не говорили об этом только что?

Да, это так. Это та же проблема. И такое же решение. Но несколько другой масштаб.

***Правило**: Сливайте код из trunk-а в вашу рабочую ветвь каждый день*



Каждый день, когда я приступаю к работе, кто-то в моей команде отвечает за заливку последней версии из trunk-а в рабочую ветвь нашей команды (= "подхватывание" ("catching up") изменений, сделанных в trunk-е).

Если моя команда (команда A) обнаруживает конфликт кода, мы решаем это немедленно - это самая приоритетная задача! Если нам нужна помощь команды B (или того, кто написал код, конфликтующий с нашим), мы берем их и работаем вместе над решением этой проблемы. Важным является то, что моя команда отвечает за решение проблемы, и что нам нужно решить ее в нашей рабочей ветви (а не в trunk-е).

***Правило**: Решайте конфликты в ветви, которая менее стабильна*

Конечно, заливка из trunk-а может быть потерей времени, если люди не делают публикации в trunk на регулярной основе. Расхождения кода между командами A и B незаметны, до тех пор, пока кто-то не сделает публикацию в trunk. Поэтому еще одно правило:

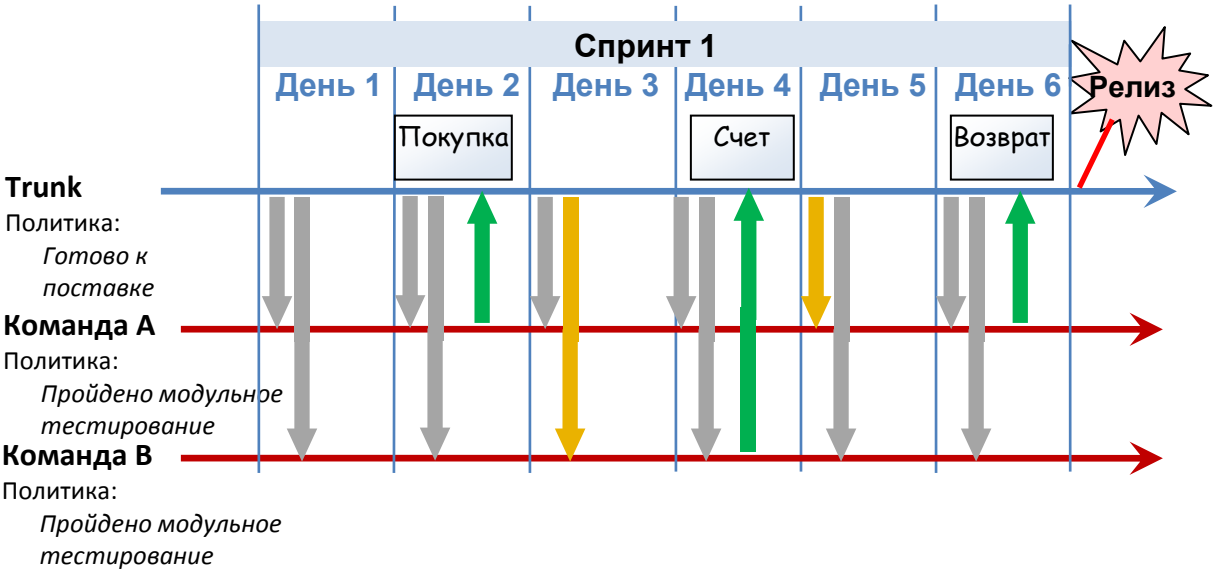
***Правило**: Заливайте код из вашей рабочей ветви в trunk на регулярной основе, например, каждый раз, когда готова пользовательская история. Не ждите до конца спринта!*

Обратите внимание на интересный побочный эффект:

***Побочный эффект:** Побеждает тот, кто первым заливает код в trunk !*

Если две команды пишут конфликтующий код, то команда заливающая код последней, должна решать конфликт. Это прекрасный побочный эффект, поскольку он заставляет команды заливать код рано :o)

Вот пример целого спринта.



Мы делаем 6-дневный спринт двумя командами. Команда А планирует реализовать истории Покупка (Book) и Возврат (Cancel). Команда В планирует реализовать истории Счет (Invoice) и Черный список должников (Blacklist). Давайте посмотрим, что произошло.

День	**Команда А**	**Команда В**	**Trunk**
1	Сливаем код в рабочую ветвь из trunk-а. Ничего нового. Работа над историей Покупка, заливка в нашу рабочую ветвь.	Сливаем код в рабочую ветвь из trunk-а. Ничего нового. Работа над историей Счет, заливка в нашу рабочую ветвь.	Сегодня ничего не произошло.
2	Сливаем код в рабочую ветвь из trunk-а. Ничего нового. Завершили реализацию истории Покупка. Прогнали интеграционные тесты. Готово! Копируем в trunk. Начинаем работу над историей Возврат.	То же, что вчера.	Пользовательская история Покупка готова!
3	Сливаем код в рабочую ветвь из trunk-а. Ничего нового. Продолжаем работу над историей Возврат.	Сливаем код в рабочую ветвь из trunk-а. Ага! Произошли изменения! Добавлена пользовательская история Покупка! Делаем слияние с нашим кодом в ветви команды В, решаем конфликты. Затем продолжаем работу над историей Счет.	Сегодня ничего не произошло.
4	То же, что вчера	Сливаем код в рабочую ветвь из trunk-а. Ничего нового. Закончили работу над историей Счет. Проводим	Готова пользовательская история Счет!

		интеграционные тесты (включая историю Покупка!), копируем в trunk. Начинаем работу над историей Черный список должников.	
5	Сливаем код в рабочую ветвь из trunk-а. Ага! Произошли изменения! Добавлен код Invoice! Делаем слияние с нашим кодом в ветви команды А, решаем конфликты. Затем продолжаем работу над историей Возврат.	Сливаем код в рабочую ветвь из trunk-а. Ничего нового. Работаем над Черным списком должников.	Сегодня ничего нового не произошло.
6	Сливаем код в рабочую ветвь из trunk-а. Ничего нового. Закончили историю Возврат, копируем в trunk.	То же, что вчера	Готова пользовательская история Возврат!

Спринт готов! Все пользовательские истории, кроме Черного списка должников, готовы. Но это нормально, мы можем делать поставку! Это возможно, потому что мы объединяли код и интегрировались постепенно. Если для этого мы ждем окончания спринта, то все расхождения в коде обнаружатся в очень неудачный момент - тогда, когда у нас меньше всего времени, чтобы исправить это.

Ветви релизов

Допустим, мы завершили спринт 1 и выпустили версию 1.0.0 системы. Сейчас, пока мы в середине спринта 2, найден серьезный дефект по выпущенной версии! О нет! Что нам делать?

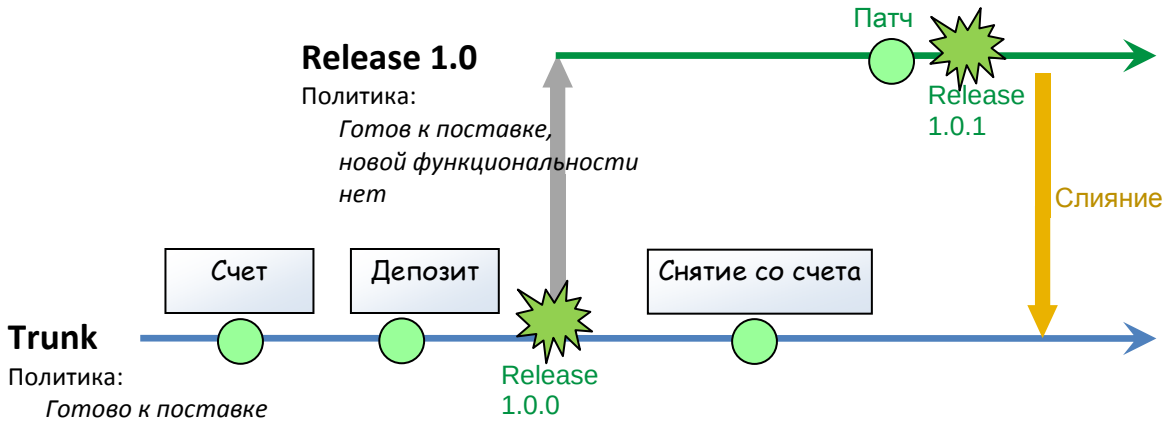
Самое простое это просто исправить этот дефект в trunk-е и выпустить версию 1.1.0. Это означает, что новые пользовательские истории, реализованные во время спринта 2 будут включены в новый релиз. Теоретически это хорошо, поскольку trunk это ветвь "Готово", и определение Готово - "готово к поставке". Поэтому, что бы ни было в trunk-е на данный момент, - это то, что мы хотим поставить.

Однако, могут быть причины, чтобы не выпускать новые пользовательские истории прямо сейчас. Например:

- Тот факт, что есть серьезный дефект, означает нарушения в trunk-е во время релиза. Что, в свою очередь означает, что пользовательские истории из спринта 2 были построены поверх разрушенной основы. Мы можем захотеть исправить основание само по себе, без новых пользовательских историй.
- Возможно заинтересованные лица (stakeholders) не хотят появления новой функциональности в середине спринта.
- Возможно, создание нового релиза из trunk-а с новой функциональностью, займет некоторое время, тогда как мы хотим сделать простой механизм, чтобы побыстрее получить исправления.

Итак, как нам сделать это?

1. Создайте *ветвь релиза* с именем release 1.0, базируясь на содержании trunk-а на момент релиза.
2. Исправьте дефект в ветви релиза.
3. Сделайте слияние изменений из ветки релиза в trunc сразу после поставки (для того, чтобы исправление попало в будущие релизы).



Обратите внимание, что нам не нужно создавать ветку release 1.0, когда мы делаем релиз 1.0.0. Мы можем ждать до появления дефектов. Т.е. нам не нужно создавать ветвь, до тех пор пока не появится что-то, что мы должны сделать в этой ветви.

Общая картина

Итак, я показал очень детальный пример использования этого шаблона на практике. Теперь, давайте немного отойдем в сторону и посмотрим на общую картину.

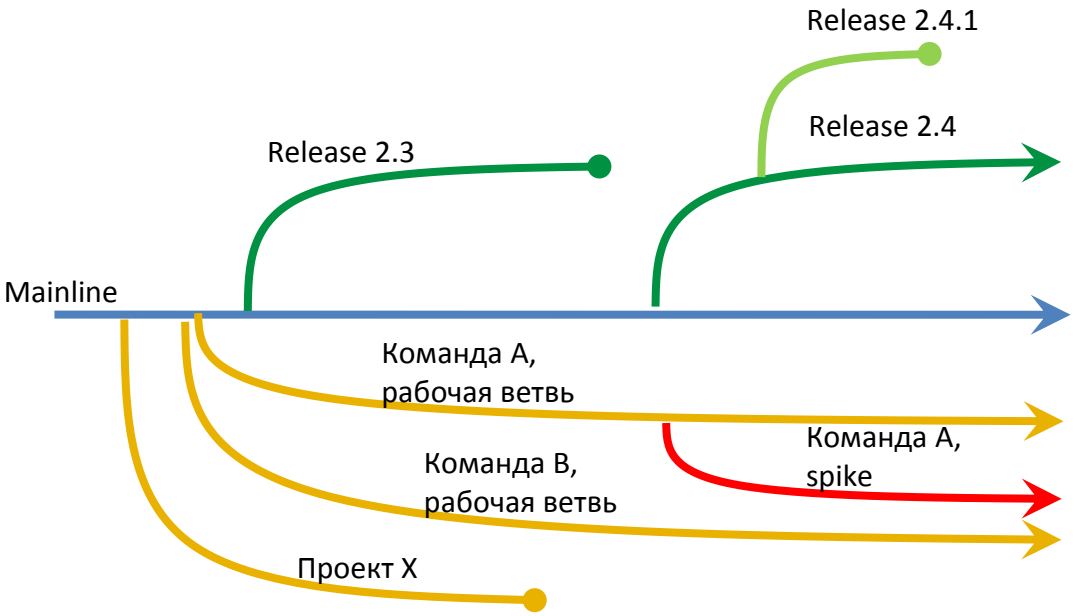
В mainline модели ветвь называется codeline (фактически, ветвь считается реализацией codeline). Иногда они называются потоками (streams).

Родитель codeline-а (т.е. codeline, от которого он произошел) называется его baseline-ом. Mainline это codeline, у которого нет baseline-а.

Из примера выше мы может сделать вывод, что:

- Trunk - это наш mainline. Ведь у него нет родителя?
- У всех остальных codeline-ов (release 1.0, рабочий для команды A, рабочий для команды B) есть trunk в качестве baseline-а.

Вот более сложный пример:



Эта картинка говорит нам, что:

- Codeline проекта X был порожден от mainline-а. Проект теперь окончен, поэтому ветвь закрыта.
- У команды A есть активная рабочая ветвь, которая была порождена от mainline-а.
- У команды A также есть активный отросток (spike), который был порожден от рабочей ветви.
- Ветвь релиза 2.3 закрыта, поскольку релиз 2.3 уже не работает в реальной среде, и поэтому его не нужно поддерживать.

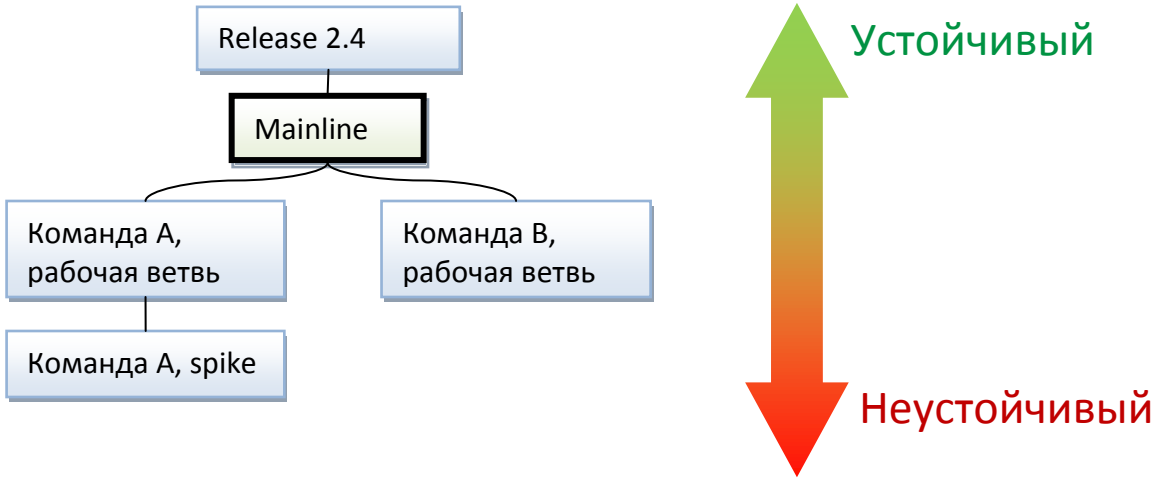
У каждого codeline-а есть есть *уровень устойчивости* относительно его baseline-а, т.е. каждый codeline либо *более устойчивый* либо *менее устойчивый* , чем его baseline.

- *Устойчивый* codeline - стабильный, тщательно протестированный, редко меняется и близок к релизу.
- *Неустойчивый* codeline - нестабильный, протестирован не полностью, часто меняется и далек от релиза.

На рисунке устойчивые codeline-ы сверху, а неустойчивые - снизу. Из приведенной выше картинки мы можем заключить, что:

- Релиз 2.3 более устойчив, чем mainline.
- Рабочая ветвь команды A менее устойчива, чем mainline.
- Отросток (spike) команды A менее устойчив, чем рабочая ветвь команды A.

Использованный выше формат рисунка полезен для представления историй ветвей, но он может быть немного путанным, когда много ветвей. Вот более ясный формат, на котором показаны только существующие в настоящий момент codeline-ы, и показано, от кого они произошли.



Я рекомендую нарисовать картинку ваших ветвей в этом формате и повесить ее на стену в комнате команды! Она действительно помогает при обсуждении интеграционных проблем.

Важное правило - все изменения движутся вдоль линий! Поэтому не нужно напрямую объединять код из рабочих ветвей команд A и B, это вызовет путанницу. Вместо этого, изменения, сделанные в рабочей ветви команды A должны перетечь в mainline , а затем далее, в рабочую ветвь команды B.

Все, что находится *выше* mainline называется *release codeline*, это означает codeline, который устойчивее, чем mainline.

Все, что находится *ниже* mainline, называется *codeline разработки* (или *рабочий*), что означает codeline , который менее устойчив, чем mainline.

***Золотое правило взаимодействия**:*
- Всегда принимайте стабилизирующие изменения
- Никогда не навязывайте дестабилизирующие изменения

Что это означает в терминах разных типов codeline-ов?

Картинка выше красочно говорит о том, что:

- Где-бы не было сделано изменение в release codeline, его нужно немедленно протащить вниз, до его baseline-а, и по всему пути вниз до mainline-а.
  - *Пример:* Исправлена ошибка в R2.4.2. Это исправление нужно немедленно протащить вниз до R2.4, и оттуда вниз до mainline -а.
- Release codeline не должен получать изменения из своего baseline -а.
  - *Пример:* Новый код внесен в mainline. Его не нужно распространять вверх до R2.3 и R2.4.
- Изменение распространяется постоянно от baseline-ов к codeline-ам разработки.
  - *Пример:* Любое изменение в mainline должно быть быстро распространено вниз до рабочих ветвей команд A и B, и от рабочей ветви команды A до отростка (spike) рабочей ветви A.
- Изменение распространяется от codeline-а разработки к его baseline-у только в стабильных точках.
  - *Пример:* Команда B заливает код из своей рабочей ветви в mainline, только когда story завершена и протестирована.

Когда делаются изменения и в codeline-е, и в его baseline-е, требуется слияние кода. Поскольку слияние кода потенциально может вызвать ошибки, нам нужно сделать это на менее устойчивом из двух codeline-ов. Когда сделали слияние кода и проверили, мы можем скопировать слитый код обратно в более устойчивый codeline.

Используя наши соглашения об изображении устойчивых codeline-ов над неустойчивыми, мы можем вывести простое правило:

***Правило**: Делаем слияние сверху вниз, копируем снизу вверх*

Пример: Команда А узнала, что в mainline-е произошли изменения. Они заливают изменения в свою ветвь разработки и разрешают конфликты. Как только ветвь разработки команды А достигла стабильной точки, они копируют код наверх, в mainline. Конечно, до этого они должны проверить, что в mainline -е больше не было новых изменений.

Вариации модели

Раздел "Шаблон управления версиями" описывает пример реализации mainline модели.

Раздел "Общая картина" дает обобщенное описание mainline модели.

В этом разделе я хочу рассказать про типичные вариации реализации этого шаблона.

"Готово" не обязательно должно означать "готово к поставке"

Просто сделайте любое определение "Готово", и выделите ветвь, в которой будут собраны пользовательские истории, которые Готовы, согласно этому определению. Будьте осторожны, однако, не включая важные вещи в понятие Готово. Если в это понятие не включено интеграционное тестирование, то когда вы будете делать интеграционные тесты?

Trunk не обязательно должен быть главной ветвью (mainline-ом)

Для, того, чтобы шаблон работал, вам нужен mainline. Но это не обязательно должен быть trunk (хотя в большинстве случаев trunk - это естественный выбор).

Командам не обязательно иметь свои собственные ветви

Несколько команд могут использовать одну ветвь, или работать непосредственно в mainline-е. До тех пор, пока соблюдена политика ветви.

Чаще командам нравится иметь собственные ветви, чтобы не мешать друг другу.

Вам не нужно каждый раз создавать новую ветвь релиза

Вы можете решить использовать ту же ветвь релиза, вместо того, чтобы создавать новую после каждого спринта. Ее можно назвать ветвь "релиза в промышленной эксплуатации" ("currently in production") или как-то в этом роде. Это подходящая модель, если у вас одновременно только одна версия в промышленной эксплуатации.

Вам не нужно делать релиз после каждого спринта

Вы можете делать релиз после каждой пользовательской истории. Или после каждого третьего спринта. Вы выбираете сами.

Вам не нужно запускать регрессионные тесты для каждой пользовательской истории

Если регрессионное тестирование или интеграция это головная боль в ваших условиях, вы можете решить оставить это до конца спринта, так что вы сможете сгруппировать несколько историй и протестировать / сынтегрировать их вместе. Это риск, который вы берете. Если регрессионное тестирование и интеграция включены в ваше определение "Готово", это просто означает, что вы рискуете получить на выходе ноль Готовых пользовательских историй, если натолкнетесь на проблемы во время регрессионного тестирования или интеграции в конце спринта.

Вопросы и ответы

Как все это согласуется с непрерывной интеграцией (continuous integration - CI)?

С какими ветвями должен работать ваш сервер непрерывной интеграции (CI сервер)? На самом деле это зависит от контекста, но вот хорошая отправная точка.

Допустим, ваша политика для trunk-а - "Готово (согласно определению) и готово к поставке", а ваша политика для каждой рабочей ветки - "модульные тесты пройдены":

- Для каждой рабочей ветви CI сервер автоматически и постоянно проверяет, что она собирается и проходит все модульные тесты.
    - Если что-то не так, он посылает сообщение об ошибке. Включается "генератор дыма" (smoke machine).
- Для каждой рабочей ветви CI сервер автоматически и регулярно (если не постоянно) запускает интеграционные и регрессионные тесты.
    - Если какой-то тест не прошел, сервер посылает предупреждение, поскольку это не входит в политику ветви.
    - Когда кто-то собирается публиковать код из рабочей ветви в trunk, этот тест запускают вручную, чтобы проверить, что пользовательская история Готова.
- Для trunk -а, CI сервер автоматически и постоянно запускает интеграционные и регрессионные тесты.
    - Если какой-то тест не прошел, сервер посылает сообщение об ошибке. Включается "генератор дыма", сирена, USB реактивная установка и вызывается национальная гвардия.

Какой инструмент лучше подойдет для этой модели управления версиями?

Точно сказать не могу. Я знаю, что это работает с Perforce, я думаю это работает с subversion, но у меня нет полной уверенности насчет CVS. Ваш вклад на этот вопрос приветствуется.

Помните, однако, одну важную вещь - цена перехода на другой инструмент обычно намного ниже по сравнению с ценой отсутствия возможности эффективного взаимодействия! Поэтому решайте, как вы хотите работать, а затем выбирайте правильный инструмент для этого.

[От переводчиков:

Сейчас популярнее децентрализованные системы контроля версий, например git. У них больше возможностей. Для git есть свой классический труд по "ветвлению": http://nvie.com/posts/a-successful-git-branching-model/ . Есть перевод на habrahabr.ru: http://habrahabr.ru/blogs/Git/106912/]

Как насчет заливок (checkins), которые не относятся к пользовательской истории?

Не все изменения кода должны относиться к пользовательской истории, я просто сделал это упрощение в своих примерах для ясности. Модель та же самая, не зависимо от типа кода, который вы заливаете (или документов).

Слияние кода всегда болезненно, поэтому я хочу делать это как можно реже!

Поздравляю, у вас merge-фобия - иррациональный страх слияния кода (merge)! Слияние кода болезненно, поскольку вы делаете это настолько редко, насколько это возможно. Чем чаще вы сливаете код, тем менее это болезненно :o)

У меня еще есть вопросы!

Посмотрите раздел Ссылки! Я уверен, вы получите ответы на большинство ваших вопросов.

Ссылки

Я настойчиво рекомендую эти ресурсы, если вы хотите почитать что-то еще по этой теме.

Practical Perforce

автор: Laura Wingerd. Вот раздел который описывает mainline модель.
Это книга по Perforce, но mainline модель специфична для Perforce.

High level best practices in Software Configuration Management

Действительно очень полезное **Обзор лучших практик управления версиями**
автор: Laura Wingerd, Christopher Seiwald.

Branching and merging - an agile perspective
Интересная статья Robert Cowham-a которая относится к нашей теме.

The Flow of Change
Отличное описание mainline модели, написанное Laura Wingerd.
Большая часть раздела "Общая картинка" основана на этих слайдах.