Управление версиями

Назначение, терминология, применение

Система управления версиями (СУВ) применяется для облегчения работы с изменяющейся информацией. СУВ позволяет хранить несколько версий одного и того же набора данных, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, создавать копии версий, объединять версии, разрешать конфликты при объединении, контролировать права доступа.

Под набором данных понимается список файлов в хранилище.

СУВ можно использовать для хранения исходных кодов ядра Домино, кодов приложений (проектов) на Домино, истории изменений приложений, документации. Но имеется важное ограничение - слияние возможно только для текстовых файлов.

При описании различных СУВ имеет место некоторая путаница терминов, поэтому дам своё понимание терминов.

Термины

- сэйв (save) - сохранение изменений в среде разработки.

  Выполняется разработчиком по мере необходимости.

- коммит (commit, revision) - фиксация изменений в СУВ.

Содержит информацию об авторе и короткое описание сути изменений.Рекомендуется делать коммит, как только появляется работоспособный прогресс.Нельзя объединять в один коммит несколько задач, поскольку СУВ рассматриваеткоммит как неделимую единицу. Коммит описывает точные различия между двумясостояниями набора данных.

- магистраль (master, origin, mainline) - главное (уникальное) направление разработки.

  Основная (эталонная) копия данных в центральном хранилище. Создаётся в начале работы и включает все те данные, которые надо обрабатывать в СУВ. Является единой точкой хранения и источником для большинства клонов.

- версия - моментальный снимок некоего набора данных.

  Версия характеризуется символической меткой. Метка описывает состояние и состав данных на момент получения снимка.

- ветвление - создание копии (клона) данных или части данных в целях отдельной разработки, проверки, эксплуатации.

  Стратегия ветвления определяется стилем взаимодействия команды разработчиков.

- ветка - копия данных, полученная в результате ветвления. Имеет идентификатор.

  Работа в пределах ветки применяется для изоляции изменений от влияния других разработчиков и задач.
  В процессе разработки в ветку записываются изменения - коммиты.

- объединение ветвей - перенос изменений (коммитов) из одной ветки в другую.

  Это может быть как однонаправленный перенос - только из одной ветки, так и слияние - совместный перенос.

Стратегия ветвлений

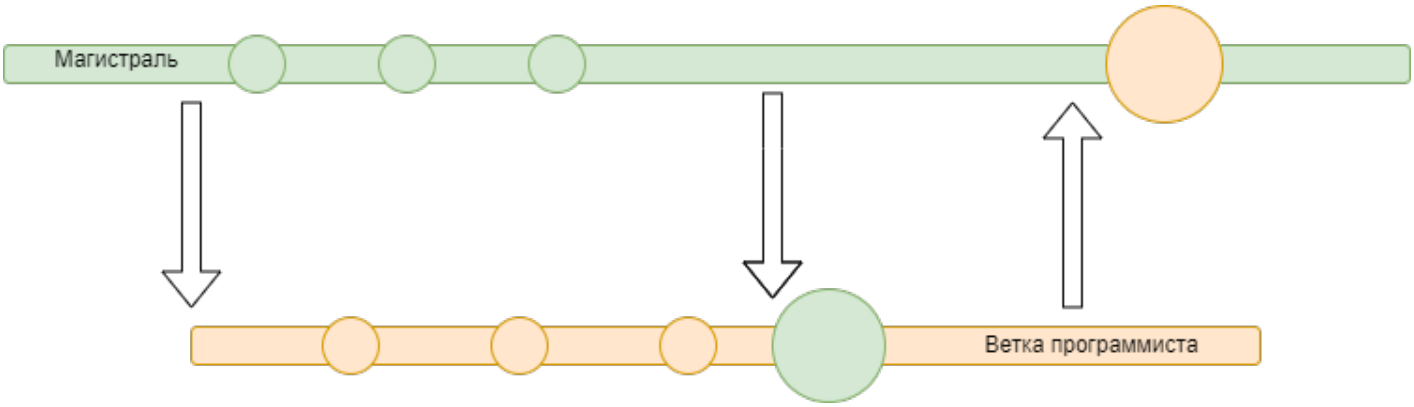
Ветвление - это поиск компромисса между изоляцией и интеграцией. Заставить всех постоянно работать над одной общей базой кода не получится. Разная скорость и качество работы программистов неизбежно вызывают конфликты. Нужно понятие частного рабочего пространства, в котором программист работает над своей задачей. В этом пространстве программист изолирован от других разработчиков. И другие программисты тоже работают в своих частных пространствах, их изменения не подвергают риску весь проект в целом. В какой-то момент потребуется интеграция, т.е. объединение результатов работы программистов. И если СУВ позволяет легко создавать ветви и отслеживать изменения в этих ветвях, то объединение происходит довольно сложно. Не существует общих критериев для автоматического объединения. Конфликтные ситуации решаются в ручном режиме.

Таким образом выбор стратегии ветвления на самом деле сводится к принятию решения о том, как и когда происходит интеграция - как часто и в каком объёме придётся решать конфликты.

Общая схема работы команды выглядит следующим образом:

1. Программист создаёт новую ветку для задачи. Предположим, что источником будет магистраль.
2. Для решения задачи вносит изменения в свою ветку
3. Выполняет тестирование на ветке. Убеждается в работоспособности программы.
4. До переноса изменений из своей ветки в магистраль следует учесть все изменения, произошедшие со времени создания ветки. Т.е. перенести новые коммиты из магистрали в свою ветку.
5. Проверяет влияние изменений (из магистрали) на работу ветки.
6. Переносит изменения из своей ветки в магистраль.

По схеме видно, что сначала выполняется изоляция (разделение на части), затем - интеграция (объединение частей).



Факторы, осложняющие интеграцию

Существуют два взгляда на суть коммита.

1. Коммит - это запись того, что уже произошло. Изменить прошлое нельзя, коммиты не изменяются.
2. Коммит - это история того, как была решена задача. Если задачу будет опубликована только после её решения, то можно изменять коммиты.

Из-за двух взглядов в СУВ имеются две команды для объединения веток.

1. слияние (merge) - слияние берет две конечные точки и сливает их вместе.
2. перебазирование (rebase) - Перебазирование повторяет изменения из одной ветки поверх другой в том порядке, в котором эти изменения были сделаны

Слияние более подвержено конфликтам, но перебазирование можно делать только с ранее неопубликованными коммитами. Иначе при повторном объединении перебазированных коммитов получится путаница.

Частота интеграции влияет на сложность объединения ветвей. Чем чаще выполняется интеграция, тем раньше обнаруживаются и быстрее разрешаются конфликты, почти исключается риск невозможного слияния

Различают следующие конфликтные ситуации в процессе объединения:

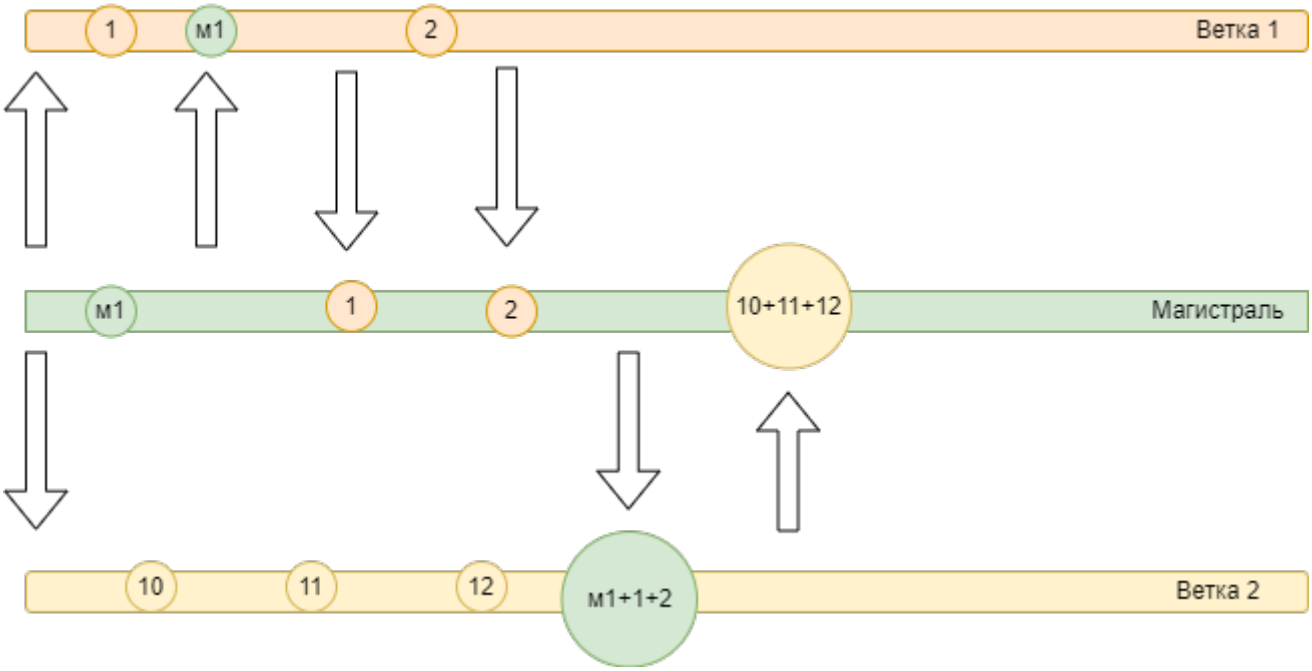
- текстовый конфликт. Например, два программиста исправили одно место. СУВ найдёт и подсветит такие места.
- семантический конфликт. Например, первый программист исправил заголовок функции, а второй вызывает эту функцию в старом виде. СУВ такие места не заметит.

Шаблоны ветвления

В литературе описаны несколько шаблонов (методик), которые позволяют командам эффективно использовать ветвление. В их основе лежат две схемы и их комбинация:

- непрерывная интеграция с магистралью
- ветвление задач и функций

Условное сравнение объёма переносимых изменений при непрерывной интеграции (ветка 1) и ветвлении по задачам (ветка 2).



Непрерывная интеграция с магистралью

Самая эффективная методика, но сложно соблюдать условия применения.

Магистраль должна находиться в стабильном рабочем состоянии.

Добиться такого состояния можно только при наличии средств самотестирования. Разрабатывается такой набор автоматизированных тестов, чтобы быть уверенным, что программа не содержит ошибок, если тесты пройдут успешно. Может быть несколько этапов тестирования. Но первый, достаточно всеобъемлющий, набор тестов, должен выполняться быстро (не более 10 минут), поскольку он будет запускаться после каждой интеграции. Если тесты завершаются неудачей, то приоритетом номер один является их исправление. Часто это означает, что ветка "замораживается" - разрешаются только исправления для восстановления работоспособности.

Схема

1. Программист создаёт новую ветку на основе магистрали.
2. Для решения задачи вносит изменения в свою ветку. Рекомендуется при каждой фиксации выполнять автоматические проверки, чтобы убедиться в отсутствии дефектов в ветке.
3. Интеграция с магистралью выполняется, как только появляется работоспособный коммит, которым можно поделиться. Не реже раза в день. Желательно, чаще.
4. Перед переносом в магистраль выполняется тестирование своей ветки. Это не только упростит привязку к магистрали, но даст уверенность, что любые ошибки, возникающие при интеграции с магистралью вызваны исключительно проблемами интеграции, а не ошибками в ветке. Это значительно ускорит и упростит поиск и исправление ошибок.
5. Выполняется перенос новых коммитов из магистрали в свою ветку.
6. Проверяется влияние изменений (из магистрали) на работу ветки.
7. Переносятся изменения из своей ветки в магистраль.

8. Запускается тестирование магистрали.

При работе в команде разработчики сливаются с магистралью последовательно, друг за другом.

Преимущества работы по данной методике:

- упрощён выпуск версий в продуктив
- сокращено время на поиск и исправление конфликтов
- быстрая интеграция
- удобнее рефакторинг кода (переработка кода программы, чтобы он стал более простым и понятным)

Условия:

- стабильно рабочее состояние магистрали
- время интеграции должно быть небольшим, иначе разработчики станут избегать слияния
- необходима автоматическая самотестируемость кода
- требуются усилия и средства для скрытия частично выполненной задачи
- достаточная квалификация членов команды. Если один член команды будет отставать, то остальные будут регулярно ждать, пока он завершит очередную интеграцию.

Ветвление задач и функций

Самая распространённая методика. На каждую задачу создаётся отдельная ветка, которая после завершения задачи интегрируется с магистралью.

Ветка задачи

Схема:

1. Разработчик создаёт ветку задачи из магистрали
2. Для решения задачи вносит изменения в свою ветку.
3. Регулярно забирает изменения из магистрали, чтобы как можно раньше обнаружить негативное влияние изменений на функциональность задачи
4. Тестирует ветку
5. Работоспособную ветку интегрирует с магистралью
6. Тестирует магистраль

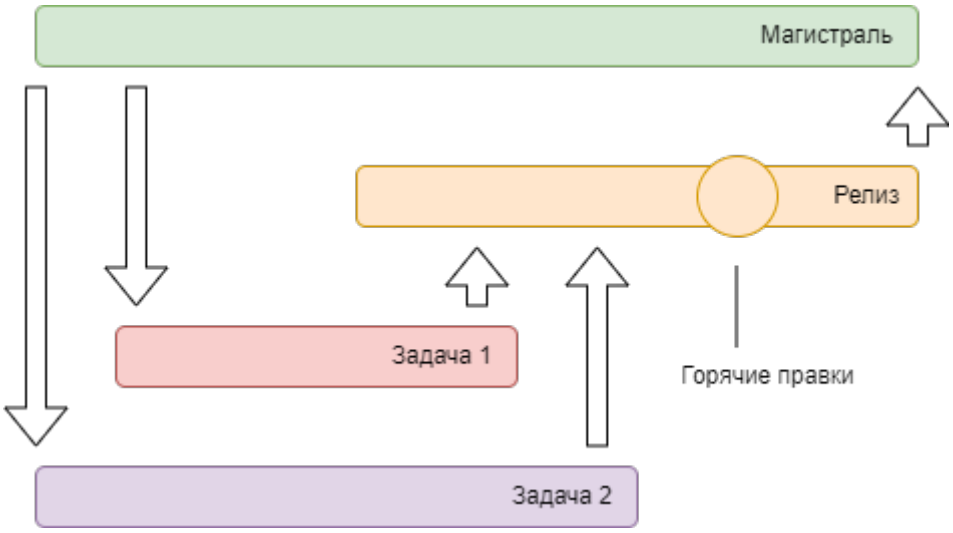
Если программист работает более чем над одной задачей одновременно, то открывает отдельную ветку для каждой из них.

Ветка релиза

Используется, если сложно поддерживать магистраль в рабочем состоянии.

Схема:

1. Ветка релиза создаётся из магистрали или из предыдущего релиза
2. Выполняется тестирование ветки релиза
3. При необходимости 'горячие правки' выполняются непосредственно в ветке релиза или в ветках, созданных из него
4. Выполняется интеграция с магистралью
5. Далее ветка релиза живёт собственной жизнью. Дальнейшие изменения в релизе могут не передаваться в магистраль.



Ветка выпуска (продуктива)

Создаётся из ветки релиза. Но чаще используется метод маркировки ветки релиза. Сначала ветке релиза назначают идентификатор: номер.test (готова к тестированию). После готовности - номер.prod (готова к выпуску).

Экспериментальная ветка

Создаётся, когда нужно что-то попробовать, но нет уверенности, что в конечном итоге это будет использовано.

Будущая ветка

Создаётся, когда необходимо внести изменения, очень сильно влияющие на базовый код, и обычные методы интеграции будут неприменимы.

Ветка совместной работы

Если требуется согласовать изменения, важные для нескольких членов команды.

Например, разработчик А создаёт процедуру, которую разработчик Б планирует использовать при решении своей задачи. Из магистрали создаётся ветка для совместной работы, а из неё ветки для каждого разработчика.

Ветка командной интеграции

Применяется при наличии нескольких команд, работающих автономно. Позволяет членам одной команды интегрироваться друг с другом без интеграции со всеми участниками

проекта.

Хранение данных

СУВ предоставляют эффективные средства для хранения данных. Это может быть как централизованное хранение (на сервере СУВ), так и распределённое.

Если нужно контролировать то, что происходит с репозиториями, то лучше всего создать собственный Git-сервер.

Существует множество систем управления версиями. Платных и без платы, с графическим и командным языками, с интеграцией с другими системами, с разными методиками и функционалом.

Git

> Командный язык

GitHub

> Предполагает высокочастотную интеграцию с магистралью (master) веток задач разработчиков. При интеграции используется механизм проверки кода другими разработчиками. Пока ветка не будет закончена интеграция не завершится.
>
> В бесплатный пакет услуг не входит хостинг приватных репозиториев.
>
> Код GitHub нельзя загрузить и развернуть на собственном сервере, поскольку код закрыт.

GitLab

> Проект с открытым кодом. Позволяет всем желающим разворачивать на собственных серверах нечто подобное GitHub. Свободно распространяемая версия GitLab имеет две редакции — бесплатную Community Edition (Core) и платную Enterprise Edition (существуют её варианты Starter, Premium и Ultimate). Последняя основана на Community Edition, которая отлично масштабируется, и, кроме того, включает в себя некоторые дополнительные возможности, ориентированные на организации. Среди возможностей GitLab можно отметить управление Git-репозиториями, средства обзора

кода, наличие системы отслеживания ошибок, ленты активности, поддержку вики-страниц. Здесь имеется и GitLab CI — система непрерывной интеграции.

Gogs

Gitea

Phabricator

GitBucket

GitFlic

Отечественная

Важные дополнения

При выборе стратегии ветвления следует учитывать следующие важные факторы.

Модульность

Модульность - это принцип построения систем, согласно которому функционально связанные части группируются в законченные узлы — модули (блоки). Модульность позволяет изменять возможности системы, путём использования/наращивания функциональных блоков, выполняющих различные задачи.

Хорошая модульность является альтернативой ветвлению, по меньшей мере, намного упрощает ветвление.

Чтобы достичь модульности, нужно постоянно следить за развитием системы и стремиться к её более модульному построению. Ключом к достижению этого является рефакторинг.

Стабильность

Многие пользователи неохотно переходят на обновление, поскольку имеют отрицательный опыт, когда обновления закончились неудачей. Однако таким клиентам по-прежнему требуются исправления ошибок и добавление функциональности. В такой ситуации команда разработчиков сохраняет открытыми ветви продукта, который все ещё используется, и применяет к ним исправления по мере необходимости. По мере разработки становится все труднее применять исправления к старым ветвям, и это становится неприятными издержками ведения бизнеса. Снизить затраты можно только поощряя клиентов к частому обновлению до последней версии. Для этого важно поддерживать стабильность продукта.

Конвейер развёртывания

Конвейер развёртывания - это процесс и технологическая инфраструктура, в которой изменения кода приложения переносятся из среды разработки в производственную среду. Процессы сборки, тестирования и развёртывания разбиваются на этапы и автоматизируются.

Рефакторинг

Это процесс изменения кода, призванный упростить его обслуживание, понимание и расширение, при этом не изменяя его назначение и поведение. Термин используется по отношению к системе в целом.

Проверка кода

Проверка кода применяется для повышения качества кода, повышения модульности, удобочитаемости и устранения дефектов в коде отдельной функции, задачи, блока. Обычно применяется перед интеграцией с магистралью.

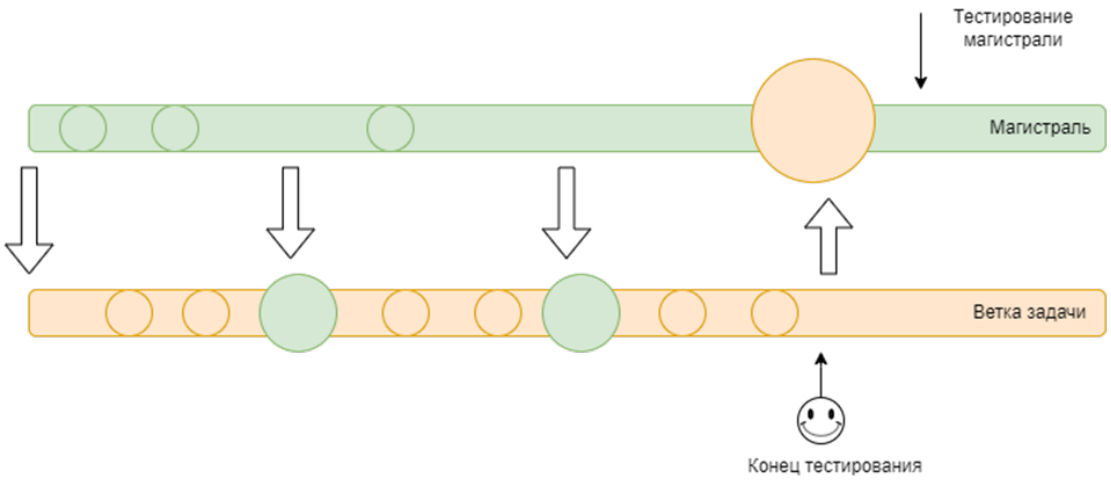
Если команда состоит из программистов разной квалификации, то слабые разработчики показывают свою работу куратору (лидеру), который проверяет их код. Может быть организована разными способами. Например, интеграция с магистралью помечается специальным состоянием коммитов. В другом варианте, разработчики сначала интегрируются в ветку куратора.

План внедрения методик GIT в Домино

Первый этап

Второй этап:

Ветвление по задачам
Под каждую задачу создаётся отдельная ветка
Задача решается внутри ветки
Проверка ветки
Перенос решения из ветки в магистраль
Проверка магистрали



Дерево приложений
Запуск с указанием ветки
Объединение сейвов в коммиты двумя способами: • простое добавление • только различия между двумя состояниями ветки
Хранение исходного кода ветки в отдельных контейнерах библиотек
Хранение коммитов в формате log-файлов Домино

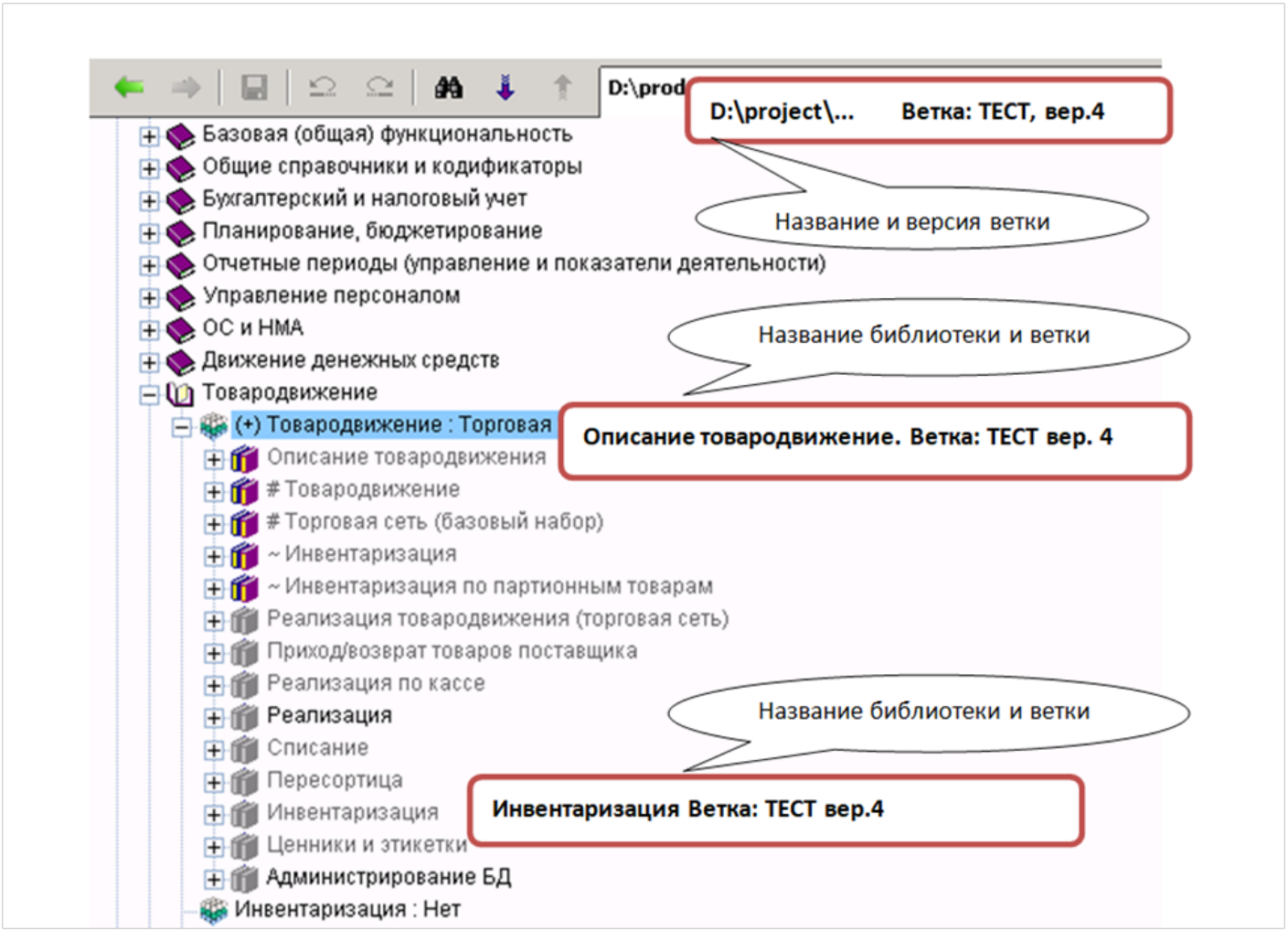
Слияние веток
• Загрузка отдельных коммитов
• Перенос различий из одной ветки в другую
• Взаимный перенос
Средство для отображения и решения конфликтов при слиянии

Git в Домино (подробнее)

Ветвление по задачам

- создание ветки (номер и комментарий, источник, дата создания, дата последнего слияния), список ветвей хранится
- изменения сохраняются в контейнере ветки. Если для библиотеки нет контейнера ветки, то он создаётся автоматически
- при запуске проекта запрашивается ветка
- в заголовке отображается номер и название ветки

Добавляем команды в сценарий: Создать ветку, Загрузить ветку



Создание коммитов в редакторе

На основании сейвов создаёт файл протокола и сохраняет его в согласованной папке. Желательно не простое объединение сейвов, а различия между двумя состояниями.

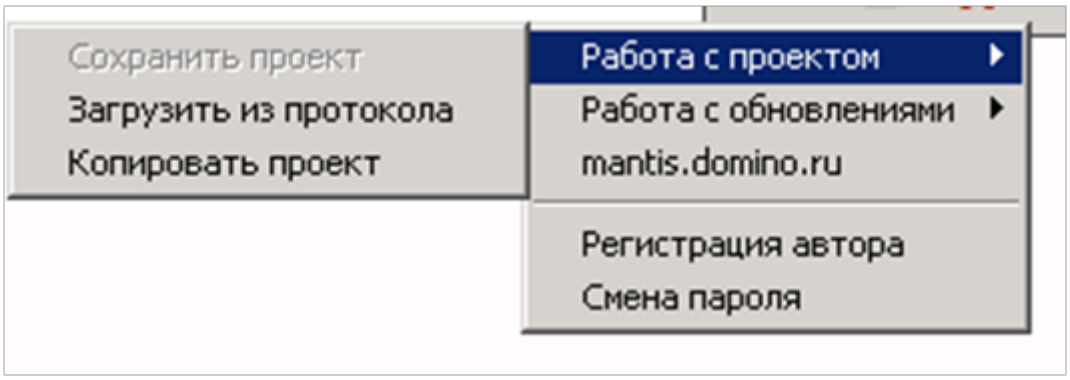
Добавляем команду в сценарий: Создать коммит с двумя подвариантами: 'Объединить сэйвы' или 'По различиям с предыдущей версией'

Файл логом коммита размещается на диске. Номер версии ветки увеличивается.

Слияние с источником

- коммитами
- перенос из источника в ветку (из ветки в источник)
- слияние (взаимный перенос)

- в случае конфликта отображаются два фрагмента дерева и предлагается выбрать итоговый вариант или ничего не делать



Добавляем команды в сценарий: Загрузить коммит, Загрузить изменения из ветки, Слияние с веткой

Автотестирование

подготовка теста, сценарий

запуск теста в автоматическом режиме

подготовка БД к тестам, восстановление до исходного состояния

сохранение результата теста

сравнение результата с эталоном в авторежиме

Сценарный тест:

- определение целей теста

- описание бизнес-процесса BPMN (**Business Process Model and Notation**)

    - выделение ролей

    - использование блоков действий

- план теста

- создание тестовых данных

- разработка и проверка

Объём тестов:

- Тестирование только ключевых процессов

- полная проверка

писать тесты так, чтобы они были повторяемы в одной и той же базе многократно

юнит-тесты или функциональные тесты

для проверки функций

создание документа

отчетная форма

Запуск разработчиками при сборке

ночные полные проверки

- функциональные
- модульные (unit)
- нагрузочные
- интеграционные
- регресионные
- smoke-тесты (дымовые тесты). Они проверяют, что система завелась и работает стабильно.

Сначала вручную составляются *тест-кейсы, сценарии и чек-листы*

Тест-кейсы - это документы, в которых прописано, что нужно проверить, какие шаги для этого предпринять и какие результаты должно показать приложение.

Атрибуты тест-кейса:
Шаги — описание последовательности действий, которые должны привести нас к ожидаемому результату. Каждый шаг отвечает на вопрос «что сделать?» (например, «зайти на страницу „Новости"», «кликнуть на кнопку „Узнать больше"»).

-

Название — основная тема тест-кейса. Краткое описание его сути.

-

•

Ожидаемый результат — то, что должно произойти после выполнения всех шагов, если функционал работает правильно.

•

Фактический результат — то, что происходит, если функционал работает некорректно (ошибка, баг).

Чек-лист

Документ, который описывает, что должно быть протестировано. Он может быть абсолютно разного уровня детализации — все зависит от требований к отчетности, уровня знания продукта сотрудниками и сложности разработки.

Баг-лист (баг-репорт)

Документ, описывающий ситуацию или последовательность действий, которые привели к некорректной работе объекта тестирования. В нем указываются причины и ожидаемый результат.

Типы автоматизированного тестирования

- Smoke Testing
- Unit Testing
- Integration Testing
- Functional Testing
- Keyword Testing
- Regression Testing
- Data Driven Testing
- Black Box Testing

MOK конструкции - без запроса данных в базе

Код тестов не входит в релиз