0) Общее описание и история развития систем контроля версий.

Изначально в разработке ПО и некоторых смежных областях для дифференциации версий информации использовали самый простой метод, при котором создавалась папка с числом, соответствующим текущей версии документа, например, diplom05\_03\_2015, или diplom\_v2.1.0.115. Это всё ещё очень популярный метод версификации, потому что он очень простой, не требует дополнительного ПО на любой современной ОС и полностью интуитивно понятный.

Тем ни менее, недостатки такого метода достаточно серьёзны: во-первых, человеку свойственны ошибки, поэтому иногда можно перепутать текущую версию документа, работать в директории, отличающейся от актуальной, а также необходимость наличия файлового хранилища большего размера для всех версий файлов.

Чтобы исправить эти ошибки, стали появляться системы контроля версий (СКВ), или VCS (Version Control System) – такие системы, которые позволяют хранить изменения в одном файле, множестве файлов или наборе папок таким образом, чтобы можно было в любой момент времени восстановить информацию до некого состояния файла или папки в прошлом, и, кроме этого, сделать это с минимально возможными временными затратами.

Лучше всего VCS работает с текстовыми файлами, так как в таком случае проще всего найти точное место, где он был изменён. Тем ни менее, хранить можно любые типы файлов; единственная проблема будет заключается в отсутствии возможности выбрать (или применить, просмотреть) какие-либо отдельные изменения, не меняя весь файл целиком.

Самой первой системой контроля версий стала SCCS – (Source Code Control System). Она использовала технику чередующихся изменений, которая теперь, в современных СКВ сменилась на версификацию и слияние.

[0] <https://ru.wikipedia.org/wiki/Source_Code_Control_System>

После этого появились RCS (Revision Control System), как более развитая альтернатива SCSS. В связи с отсутствием ПО, которое предоставляло бы схожий функционал, RCS быстро завоевали популярность не только у разработчиков, но и некоторых других категорий пользователей. К слову, RCS, созданная в 1982 году, до сих пор поставляется в стандартной сборке Developer Tools с Mac OS X. Диаграмму взаимоотношений между пользователем и VCS сервером можно увидеть на схеме 0:

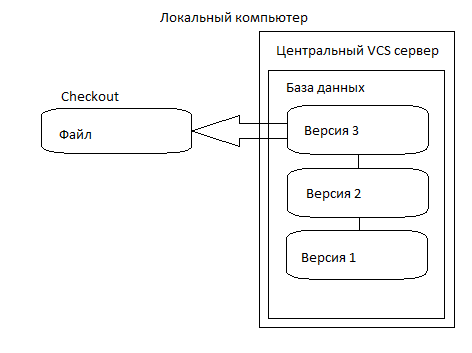


Схема 0 – Диаграмма взаимоотношений в RCS.

Тем ни менее, несмотря на относительную успешность и тот факт, что RCS по-прежнему используется в реальных проектах (TWiki), в этой системе существует масса проблем, которые и помешали дальнейшему распространению такой системы. Так, например, RCS взаимодействует только с выбранными файлами: нет возможности работать над все проектом целиком. Кроме того, несмотря на наличие функции разветвления (branching), синтаксис поведения очень тяжёл для понимания. Поэтому многие команды использовали встроенный механизм блокировки и работают в одной ветке head. На практике это означает, что только один пользователь может менять какой-либо файл в данный момент (lock-unlock механизм); пока он не закончит свою работу над ним, никто не сможет изменить данный файл.

[1] <http://en.wikipedia.org/wiki/Revision_Control_System#Mode_of_operation>

1) Обзор централизованных и децентрализованных систем контроля версий.

С приходом более совершенных IDE и языков программирования, вопрос взаимодействия с коллегами является ключевым для многих разработчиков. В основном чтобы решить эту проблему, начали появляться централизованные системы контроля версий (ЦСВ или CVS – Centralized Version Control systems). Примерами таких систем являются:

* CVS;
* Subversion;
* Perforce.

В них используется следующий метод сохранения информации: имеется один сервер, который хранит все версии файлов, а также некоторое количество клиентов, на которые поставляется информация с сервера. Диаграмма таких взаимоотношений представлена на схеме 1:



Схема 1 – Диаграмма взаимоотношений в VCS

Такой подход имеет много преимуществ, особенно относительно RCS. Основное из них в том, что разработчики могут работать с одним и тем же файлом одновременно. Кроме этого, есть ещё и другие плюсы, например, каждый разработчик в любой момент времени может знать, что делают все остальные; кроме этого, администраторы систем, имея точно такую же возможность, могут эффективнее управлять своими кадрами и всем продуктом в целом.

Но, несмотря на решение достаточно большого спектра проблем, у таких систем есть свои недостатки. Первый и наиболее существенный – единая точка отказа. Если сервер по каким-то причинам выходит из строя, то все работают только со своими локальными копиями, не имея возможности смотреть изменения других разработчиков.

Чтобы решить эту проблему, и, в свою очередь забыть о проблемах RCC, существуют децентрализованные системы контроля версий (ДСКВ или DVCS – Distributed Version Control System). В качестве примеров можно привести следующие ДСКВ:

* Git;
* Mercurial;
* Darcs;
* Bazaar.

Как следует из названия, в таких системах клиенты не просто забирают последнюю (или выбранную, но всегда одну) версию проекта: на самом деле они полностью копируют его содержание и историю. Таким образом, при таком подходе у системы появляются два больших преимущества перед VCS:

* В любой момент времени разработчик может посмотреть состояние проекта в какой-то момент времени даже без наличия доступа к центральному серверу;
* Каждая копия проекта это, по сути дела, полная резервная копия проекта. Таким образом, если сервер выйдет из строя, то любой разработчик сможет восстановить его состояние на любой другой машине.

Диаграмма взаимоотношений между клиентом и сервером представлена на схеме 2:



Схема 2 – Децентрализованная система контроля версий

1.1 Примеры реализаций и разница между ними.

2) Типовые операции и порядок работы с системой контроля версий.  
Если не вдаваться в подробности реализации и работы с системами контроля версий, то общий порядок работы примерно одинаков для любой VCS.

Первое, что делает любой разработчик, начиная работать с проектом – создание рабочей копии проекта. В Subversion для этих целей служит команда checkout, для Git – clone. После выполнения этой команды на рабочей машине разработчика появляется директория, в которой находятся непосредственно сам проект. После этого можно непосредственно приступать к разработке.

Стоит отметить, что, в зависимости от реализации системы контроля версий, может установиться не одна папка, а две: одна с рабочей копией, а вторая – полный её дубль, позволяющий в любой момент времени локально определить, какие файлы были изменены.

Далее разработчик будет модифицировать рабочую копию директории, чтобы в дальнейшем зафиксировать изменения (commit). После этого изменённое состояние отправляется на сервер (push). Кроме этого, можно узнать текущее состояние сервера (pull).

Тем ни менее, такой идеальный сценарий случается редко, так как клиенты зачастую вносят правки в одну и ту же копию документа одновременно (при этом два и более клиента могут быть одним и тем же пользователем, просто работающим с разных машин). В таком случае применяется система разрешения конфликтов (resolve conflicts). Существуют следующие способы решения конфликтов:

* Вручную. При таком сценарии каждый файл пересматривается руками; из него удаляются вспомогательные строки, например, “<<<<<<< HEAD” и “>>>>>>> master”, указывающие пользователю, каким образом строки конфликтуют между собой, а также применяются изменения, которые должны быть зафиксированы в текущей версии проекта;
* При помощи специальных программ. На данный момент их насчитывается несколько десятков, самые популярные из них это KDiff3, DiffMerge, P4Merge и другие, которые могут работать на разных ОС, с разными СКВ, с разными проектами. Кроме этого, большинство таких систем бесплатны или условно бесплатны, последняя даёт возможность пользования в целях учёбы или в некоммерческих организациях;

[2] <http://stackoverflow.com/questions/572237/whats-the-best-three-way-merge-tool>

* При помощи встроенных возможностей СКВ. Некоторые СКВ имеют возможность автоматического разрешения конфликтов, в более же сложных ситуациях же они отдают выбор способа решения пользователю (например, команда git mergetool).

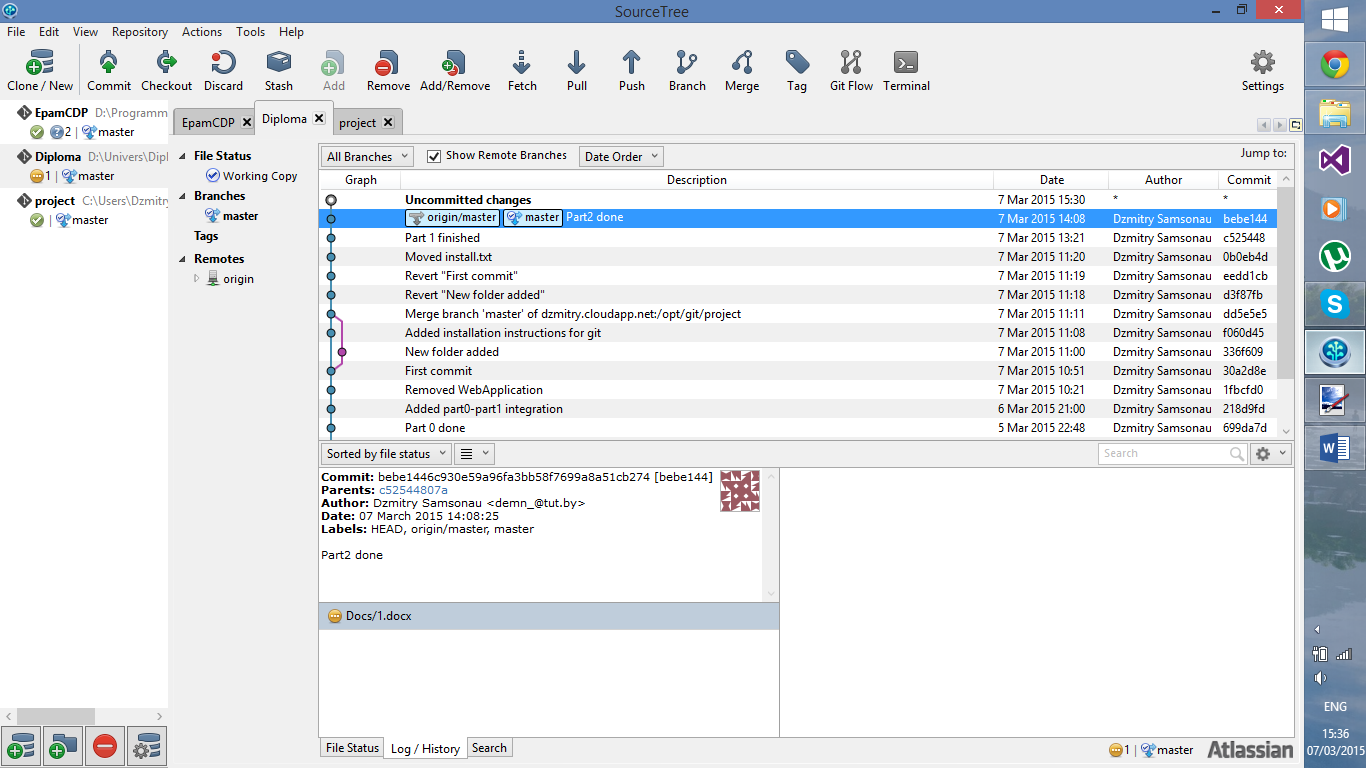
Чаще всего такие ситуации возникают при слиянии двух веток в одну, так как традиционно процесс разработки в команде строится таким образом, чтобы каждый разработчик работал в одной ветке проекта, а потом, после выполнения текущего задания и удачного прохождения тестов (или после ручной проверки), объединял все изменения в главную ветку (чаще всего – master).

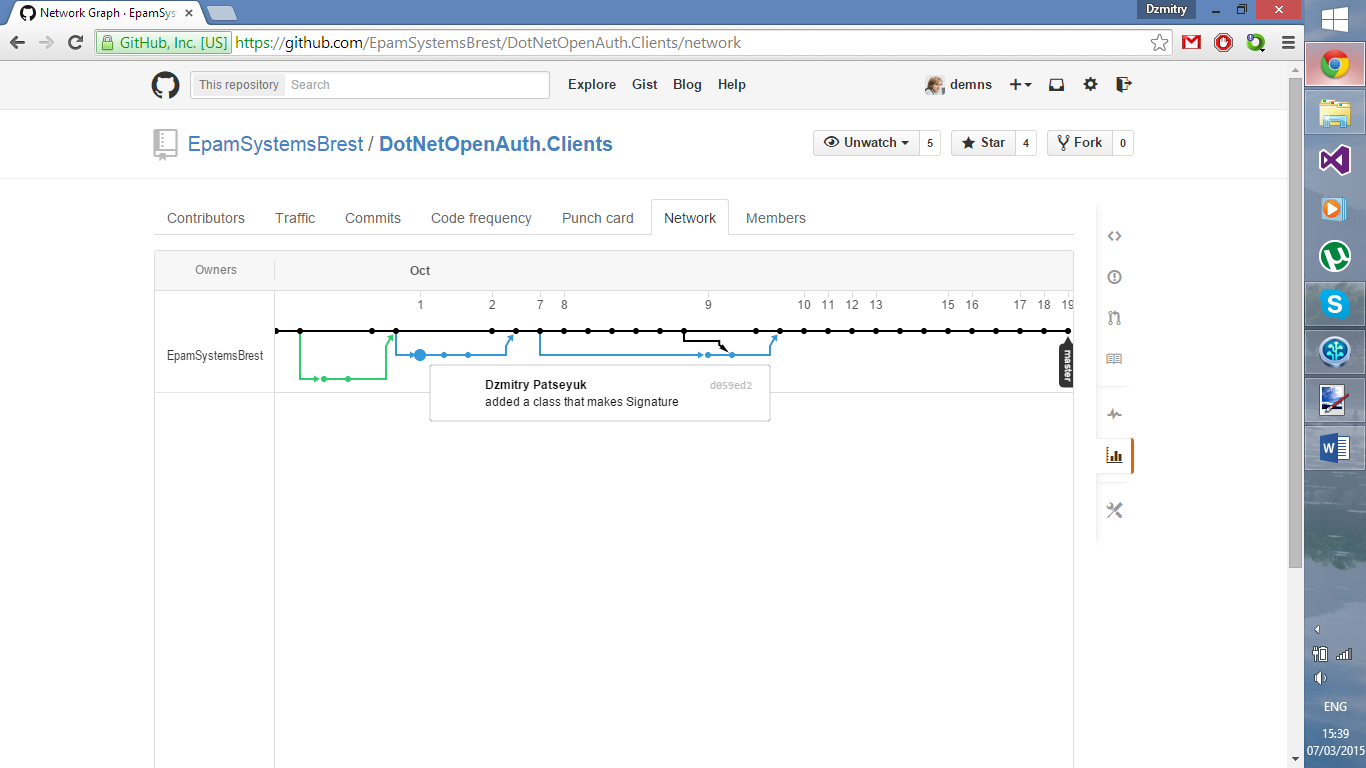
Ветка (branch) – это на самом деле ссылка на какую-либо версию проекта. Каждый раз, когда происходит фиксация изменений, ссылка автоматически меняет свой адрес на следующую версию. Именно при помощи механизма веток в VCS стало возможным распараллеливание работы разработчиков.

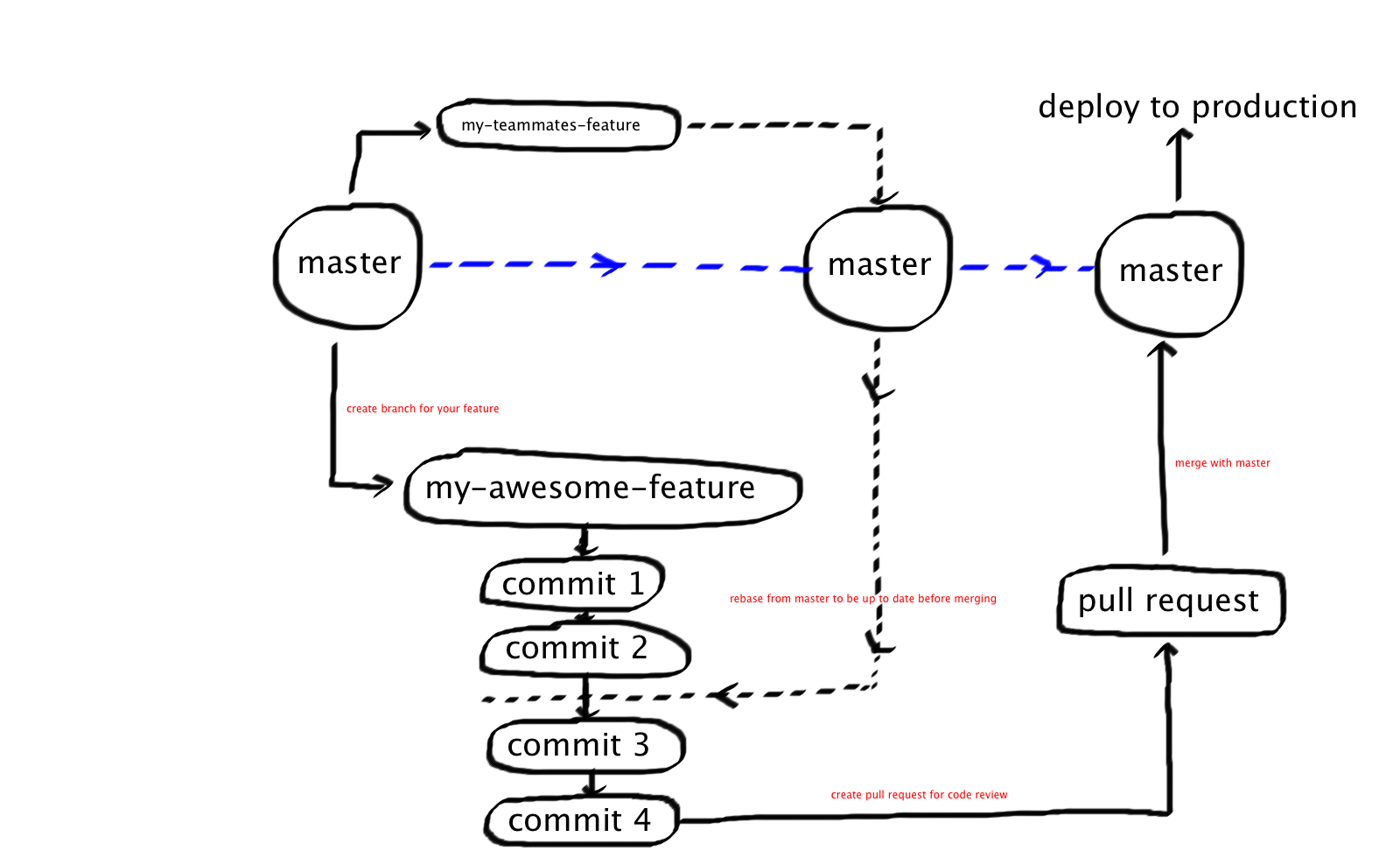
2.1 Основные отличия операций CVS и DVCS.

2.2 Продвинутые операции (cherry-pick, reverse, stash, etc).

Спросить про картинки:







3) Разработка концепции лабораторного практикума.

Основная цель работы – разработать лабораторный практикум, благодаря которому можно было бы рассмотреть основы централизованных и децентрализованных систем, рассмотреть их преимущества и недостатки, а также попробовать свои силы при создании нового проекта.

Таким образом, в практикуме можно выделить две части: теоретическую и практическую. В теоретической части необходимо будет сделать разделение на 4 лабораторные работы: (здесь я не знаю, какие они будут, равное ли разделение, нужны ли advanced technics).

Практическая часть, в свою очередь, делится на клиентскую и серверную сторону. На серверной стороне необходимо установить Git и SVN системы, для того, чтобы клиенты смогли с ними работать. У учащихся не подразумевается наличие какого-либо дополнительного оборудования и ПО, кроме того, что будет установлено в процессе выполнения лабораторных работ; у преподавателя должна иметься возможность быстро проверить работы (а ещё лучше – оценивать работы автоматом).

Установка Git на сервер (Ubuntu 12.04) выглядит следующим образом:

1. Создаётся git-пользователь:
   1. $ sudo adduser git
   2. $ su git
   3. $ cd
   4. $ mkdir .ssh
2. Создаётся ssh-ключ для каждого пользователя:
   1. $ cat /tmp/id\_rsa.john.pub
   2. #повторить для всех пользователей
3. Добавить ключи в .ssh директорию:
   1. $ cat /tmp/id\_rsa.jessica.pub >> ~/.ssh/authorized\_keys
   2. $ cat /tmp/id\_rsa.john.pub >> ~/.ssh/authorized\_keys
   3. $ cat /tmp/id\_rsa.josie.pub >> ~/.ssh/authorized\_keys
4. Изменить security-настройки:
   1. chmod -R go= ~/.ssh
5. Создать пустой git-репозиторий:
   1. $ cd /opt/git
   2. $ mkdir project.git
   3. $ cd project.git
   4. $ git --bare init
6. Убрать shell для git-пользователя:
   1. $ sudo vim /etc/passwd
   2. Поменять строку /bin/sh на /usr/bin/git-shell

[4] https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-git-server-on-a-vps

[5] http://stackoverflow.com/questions/6448242/git-push-error-insufficient-permission-for-adding-an-object-to-repository-datab

[6] http://git-scm.com/book/en/v1/Git-on-the-Server-Setting-Up-the-Server

Дальнейшая работа производится учащимися на своих компьютерах при помощи следующих настроек:

cd git\_repo

git init

type > 1.txt

git add .

git commit -m 'First commit'

git remote add origin git@dzmitry.cloudapp.net:/opt/git/project.git

????????????????????????????????????????