Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования
«Севастопольский государственный университет»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторным работам по дисциплине

"Технологии создания программных продуктов"

Часть 2

для студентов, обучающихся по направлению

09.03.02 “Информационные системы и технологии”

очной и заочной форм обучения

Севастополь
2015

УДК 004.732

Методические указания к лабораторным занятиям по дисциплине "Технологии создания программных продуктов". Часть 2 / Сост. Строганов В.А., Дрозин А.Ю. — Севастополь: Изд-во СевГУ, 2015— 40 с.

Методические указания предназначены для проведения лабораторных работ по дисциплине “Технологии создания программных продуктов“. Целью методических указаний является помощь студентом в выполнении лабораторных работ. Излагаются теоретические и практические сведения необходимые для выполнения лабораторной работы, программы работы, требования к содержанию отчета. Целью лабораторного практикума является приобретение студентами практических навыков составления диаграмм на языке UML с использованием соответствующих CASE-средств, применения диаграмм при разработке программных систем.

Методические указания рассмотрены и утверждены на методическом семинаре и заседании кафедры информационных систем

(протокол № 1 от 31 августа 2015 г.)

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Рецензент: Кротов К.В., канд. техн. наук, доцент кафедры ИС

Содержание

Лабораторная работа №1

Исследование распределенных систем контроля версий Mercurial при коллективной разработке программных продуктов

1. Цель работы

Исследовать основные подходы к организации взаимодействия команды разработчиков с использованием распределенной системы контроля версий (DVCS). Приобрести практические навыки установки и настойки DVCS Mercurial, организации ветвей разработки и осуществление слияния.

2. Основные положения

2.1. Общие принципы организации централизованных и распределенных системы контроля версий

В классических централизованных системах контроля версий (Subversion, CVS) — есть выделенное специальное хранилище называемое репозиторий, в котором хранится программный код некоторого проекта и вся история изменений. И вот к этому хранилищу обращаются попеременно все работающие над проектом.

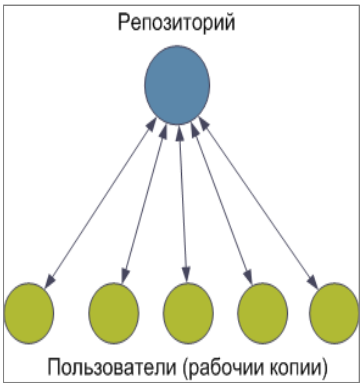

Рисунок 2.1 — Модель централизованной системы контроля версий

При использовании данной модели возникает целый ряд проблем, связанных с тем, что все разработчики вынуждены работать с одним, общим репозиторием. При этом главная проблема, к которой постепенно приходят все группы разработчиков — это то что, в больших командах возможно вносить изменения только большими частями кода, которые покрыты тестами и могут уже использоваться. Тому много причин, но главное — страх поломать что-то готовое в репозитирии, что кем-то используется. Где разработчикам хранить промежуточные изменения не совсем понятно. Механизм так называемых ветвей в SVN реализован достаточно сложно и не может считаться приемлемым решением проблемы.

Еще одной важной проблемой является чрезмерная перегруженность сервера, на котором работает централизованный репозиторий. Выход этого сервера из строя приводит к катастрофическим последствиям.

Для решения указанных выше проблем была предложена более сложная концепция — распределенные системы контроля версий (DVCS). У каждого пользователя при этом есть свой локальный репозиторий, причем вовсе не обязательно один. Централизованный репозиторий отсутствует.

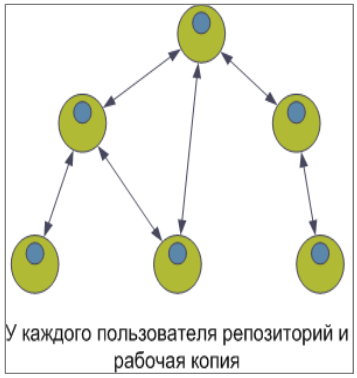

Рисунок 2.2 — Модель распределенной системы контроля версий

За счет локальности достигается большая гранулярность — теперь можно вносить изменения в репозиторий не опасаясь поломать чужой код, да и весь проект, при этом вы всегда знаете, что история сохраняется, даже в том случае если вы не имеете доступа к основному репозиторию, например, в случае отсутствия доступа в интернет.

Понятие основного репозитория в случае распределенных систем контроля довольно условное. Он основной, потому что некто его так назвал. Ничто не мешает вам взять и забрать обновления лично у программиста X, а ему у вас, да и отправить свои обновления другому — тоже тривиальная задача. Естественно если это позволяют настройки прав доступа. Таким образом, получаем, что в распределенных системах отсутствует строгая иерархичность — все репозитории равны, и рядом с каждым репозиторием может быть размещена собственная рабочая копия, хотя и не обязательно.

Смотря на такую структуру, возможность локальных изменений, возможность синхронизации состояния репозитория с кем угодно создается ощущение, что исходные тексты проекта превратятся в кашу, и на определенном этапе, причем совсем недалеком от начала, уже невозможно будет как-то получить адекватное их состояние. На самом деле все не так страшно. Мощнейшей вещью распределенных систем контроля версий — является ветвление. В DVCS, ну по крайней мере в Mercurial, ветвление — это повседневная операция, это в принципе основа контроля версий в данном случае. Реализована она абсолютно логично и понятно, и действительно проста в использовании.

Однако, как считают некоторые эксперты, в распределенных системах контроля версий их распределенность не является самой интересной особенностью. Наиболее интересным является изменение модели —

распределенные системы контроля версий работают с изменениями (changes), а не с версиями. Если централизованная система контроля версий «думает»: у меня есть версия 1, после этого будет версия 2, после этого версия 3 и так далее. В распределенной системе все по другому: сначала не было ничего, потом добавлены эти изменения, потом добавлены те, и т.д. Изменение программной модели должно изменить модель пользователя. Теперь разработчикам тоже необходимо мыслить в терминах изменений. Если раньше было: «Я хочу получить версию номер X», или «Я хочу последнюю версию», то теперь: «Хочу получить набор изменений программиста X.».

Только когда разработчики начнут мыслить в терминах «изменении», и выбросите из головы «версии» все встанет на свои места. Именно изменение модели работы системы контроля версий привело к существенному упрощению слияния (`merge`) кода. И соответственно к более активному использованию ветвления, использованию его там, где оно необходимо. Теперь нет необходимости думать о сложностях последующего слияния создавать долгоживущие ветви для команд тестирования и поддержки и создавать короткоживущие ветви для экспериментов.

2.2. Основы работы в Mercurial

Центральным понятием Mercurial является ***ревизия***, которая здесь называется ***changeset.*** В связи со спецификой распределенных систем контроля версий невозможно выдать каждой ревизии её номер, поскольку не получится гарантировать его уникальность среди всех существующих репозиториев. Однако каждая ревизия все-таки имеет уникальный идентификатор, в случае Mercurial это 40-значный sha1-хеш, который учитывает все параметры ревизии. Таким образом, у каждой новой ревизии в любом удаленном репозитории будет свой уникальный идентификатор. Использование подобной нумерации ревизий немного пугает начинающих пользователей, однако ничего страшного в них нет, и использование тех или иных идентификаторов это просто дело привычки.

Вся работа с системой контроля версий Mercurial происходит с помощью команды `hg`, и во всех разделах далее будут приводиться именно консольные команды, и консольные способы работы.

Работа с этой системой контроля версий, как, впрочем, и со всеми остальными, начинается с создания репозитория в пустом каталоге файловой системы. Для этого следует перейти в выбранный каталог, например, `~/repos/hgproject`, и выполним команду:

```
> hg init
```

По команде `hg init` Mercurial создает репозиторий в текущем каталоге. Если посмотреть на результат работы, то можно увидеть каталог `.hg`, в которой собственно и хранится вся история работы над проектом.

Далее следует создать некое подобие обычной структуры работы над проектом. Для этого надо создать каталог, в котором будет располагаться проект и перейти в него, пусть это будет `~/projects`.

Теперь нужно получить данные для начала работы над проектом. В общем случае это будет все содержимое некоторого репозитория расположенного где-то на сервере. Для этого перейдем в `~/projects` и выполним команду:

```
> hg clone ~/repos/hgproject
```

По команде `hg clone` Mercurial «клонирует» репозиторий расположенный по указанному адресу в текущий каталог. При этом к вам попадает именно репозиторий, то есть хранилище, содержащее всю существующую историю изменений. Таким образом, уже появляется два репозитория — то есть локально получена распределенная система контроля версий. Взаимодействие может происходить с любым имеющимся репозиторием, так как они все равноценны, однако будем называть репозиторий в каталоге `~/repos/hgproject` "центральным", то есть введем конвенцию на взаимодействие с системой.

С помощью текстового редактора необходимо создать новый файл в каталоге с проектом, пусть для примера это будет readme.txt, и напишем строку символов в этот файл. Таким образом, получен файл в проекте, который необходимо хранить в репозитории. Перед тем, как сохранить новый файл в репозитории следует убедиться в том, что Mercurial его видит, для этого в каталоге с новым файлом необходимо выполнить:

```
>hg status
? readme.txt
```

Mercurial ответил, что он видит файл readme.txt, при этом этот файл пока не находится в системе контроля версий (символ «?» слева от имени файла). По команде `status` Mercurial выводит состояние рабочей копии в сравнении с состоянием локального репозитория. Для того, чтобы сказать Mercurial, что его необходимо версионировать выполним:

```
> hg add
adding readme.txt
```

И ещё раз:

```
> hg status
A readme.txt
```

Слева от имени файла появился символ «A», который означает что файл readme.txt будет добавлен в систему контроля версий при следующем выполнении команды `commit`, которая как бы вносит появившиеся изменения в репозиторий или так сказать, подтверждает и фиксирует их там.

```
>hg commit
```

Mercurial запустит текстовый редактор и попросит ввести описание к вносимым изменениям. Как только редактор будет закрыт, все изменения в рабочей копии будут сохранены в локальном репозитории. Убедиться в этом достаточно просто:

```
>hg log
changeset:   0:8fae369766e9
tag:         tip
user:        mike@mike-notebook
date:        Fri Nov 27 08:58:01 2009 +0300
summary:     Файл readme.txt добавлен в репозиторий
```

`Changeset` — это и есть номер ревизии, который состоит из двух частей: виртуального номера ревизии (записан до «:») и идентификатора (sha1-хеша). Виртуальный номер ревизии призван облегчить жизнь пользователям, и все-таки ввести в эту систему некоторую нумерацию ревизий. Но, как показывает практика использовать этот номер для однозначной идентификации нельзя, так как может привести к путанице в понимании происходящего в репозиториях. Обычно для однозначной идентификации версии достаточно 4-5 шестнадцатеричных цифр идентификатора. Следующей строкой идёт «`tag: tip`», в общем `tip` — это обозначение последней ревизии, хотя выбирается это обозначение в различных случаях по различным принципам, далее, при рассмотрении организаций ветвлений этот момент будет исследован более подробно. Значение следующих строк очевидно, и нет необходимости их как-либо комментировать.

Выполнение команды `commit` локально, то есть история изменений сохранены только в данном локальном репозитории. Для того, чтобы передать изменения в репозиторий расположенный в `~/repos/hgproject` следует выполнить:

```
> hg push
pushing to ~/repos/hgproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

После выполнения этой команды все изменения, зафиксированные в локальном репозитории, были зафиксированы также и в удаленном.

Теперь следует склонировать репозиторий ещё раз, и посмотреть как происходит обмен ревизиями в Mercurial. Для этого необходимо создать новый каталог `~/projects/hgproj_clone`, и склонировать в него наш удаленный репозиторий:

```
>hg clone ~/repos/hgproject ~/projects/hgproj_clone
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

И уже во вновь склонированном репозитории создадим файл other.txt с помощью текстового редактора. И снова повторим операции описанные выше:

```
> hg status
? other.txt
> hg add
adding other.txt
> hg commit
> hg log
changeset:   1:270e49e72f4b
tag:         tip
user:        mike@mike-notebook
date:        Fri Nov 27 10:39:35 2009 +0300
summary:         Записан файл other.txt в другом репозитории

changeset:   0:8fae369766e9
user:        mike@mike-notebook
date:        Fri Nov 27 08:58:01 2009 +0300
summary:     Файл readme.txt добавлен в репозиторий
```

Видим, что в новом репозитории отражены как изменения, сделанные локально, так и изменения, сделанные в удаленном репозитории, которые мы ранее отправляли командой `hg push`. Теперь необходимо воспользоваться еще одной командой:

```
> hg outgoing
comparing with ~/repos/hgproject
searching for changes
changeset:   1:270e49e72f4b
tag:         tip
user:        mike@mike-notebook
date:        Fri Nov 27 10:39:35 2009 +0300
summary:         Записан файл other.txt в другом репозитории
```

По команде `hg outgoing` Mercurial выводит список ревизий, которые есть в вашем локальном репозитории, но которых нет в «центральном». Отправить появившиеся ревизии в «центральный» репозиторий можно рассмотренным ранее способом:

```
> hg push
pushing to ~/repos/hgproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Таким образом, в «центральном репозитории две ревизии. Теперь рассмотрим, как следует забирать обновления из центрального репозитория. Для этого перейдём в каталог с первым клоном, то есть в `~/projects/hgproject`, и выполним:

```
> hg incoming
comparing with ~/repos/hgproject
searching for changes
changeset:   1:270e49e72f4b
tag:         tip
user:        mike@mike-notebook
date:        Fri Nov 27 10:39:35 2009 +0300
summary:         Записан файл other.txt в другом
репозитории
```

Команда `hg incoming` выдает список ревизий, которые есть в удаленном репозитории, но отсутствуют в локальном. А затем можно получить эти ревизии, для чего надо выполнить:

```
> hg pull
pulling from ~/repos/hgproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
(run 'hg update' to get a working copy)
```

Команда `hg pull` получает ревизии из удаленного репозитория, и добавляет их в локальный, таким образом, изменения из «центрального» репозитория были перемещены в локальный репозиторий. Но они остались только в репозитории, локальная копия осталась нетронутой. Для того, чтобы обновить локальную копию выполним:

```
> hg update
```

```
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Если посмотреть на состояние рабочей копии, то она соответствует состоянию рабочей копии в репозитории `~/projects/hgproj_clone`, а состояние хранилища во всех трех репозиториях одинаково.

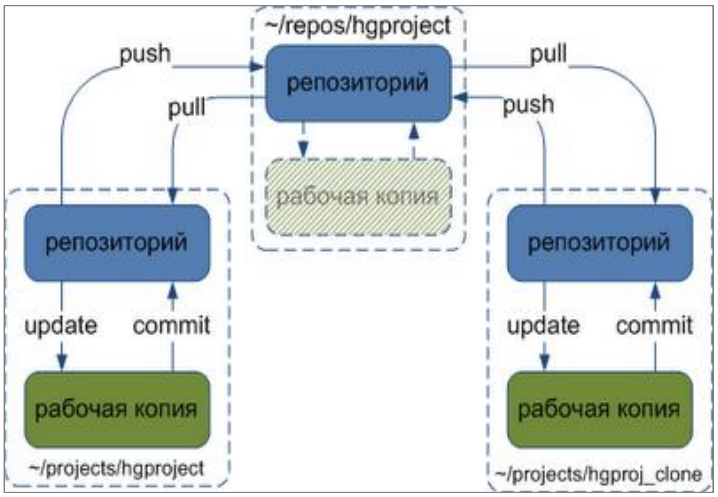

Рисунок 2.3 — Основные команды работы в Mercurial

2.3 Работа с ветвями и слияниями в Mercurial

В данном разделе будут рассмотрены сложные операции с репозиториями, а именно — создание ветвей и работа с ними, а также сопутствующие вопросы

Первым ветвлением, с которым столкнется команда, работающая с Mercurial — это ветвление при помещении в центральный репозиторий новых изменений. Ситуация возникает когда в локальном репозитории имеются изменения подтвержденные командой `commit`, но не отправленные в центральные репозиторий командой, и в тоже время один (а то и несколько) коллег поместили в "центральный" репозиторий свои изменения. Далее поясним ситуацию на описанных в предыдущем разделе трех репозиториях. Итак, в обоих репозиториях сохранено по две ревизии, при этом оба репозитория были синхронизированы с "центральным". Внесем в репозитории различные изменения и рассмотрим, что из этого выйдет.

Для начала необходимо создать файл first.txt в первом репозитории, подтвердим добавление его и отправим изменения в центральный репозиторий:

```
> echo "new text to first.txt" > first.txt
> hg status
? first.txt
> hg add first.txt
> hg commit
> hg outgoing
```

```
    comparing with /home/mike/Repositories/newProject
    searching for changes
    changeset:   2:66c5686e355e
    tag:         tip
    user:        mike@mike-vbox
    date:        Thu Jan 07 22:28:39 2010 +0300
    summary:         Коммит файла first.txt в первом
репозитории
    > hg push
    pushing to /home/mike/Repositories/newProject
    searching for changes
    adding changesets
    adding manifests
    adding file changes
    added 1 changesets with 1 changes to 1 files
```

А теперь с эмулируем ситуацию, когда коллега также внес изменения, отличающиеся от рассмотренных выше, и посмотрим как такая ситуация решается средствами Mercurial, ведь подобная ситуация в случае командной разработки будет достаточно частой. Для этого переместимся в имеющийся у нас второй репозиторий, создадим в нем новый файл, и посмотрим результат:

```
    > echo "file created in second repository" >
second.txt
    > hg status
    ? second.txt
    > hg add
    adding second.txt
    > hg commit
    > hg log
    changeset:   2:6872fa960507
    tag:         tip
    user:        mike@mike-vbox
    date:        Sun Jan 10 19:40:45 2010 +0300
    summary:         Файл second.txt создан во втором
репозитории

    changeset:   1:270e49e72f4b
    user:        mike@mike-notebook
    date:        Fri Nov 27 10:39:35 2009 +0300
    summary:         Записан файл other.txt в другом
репозитории

    changeset:   0:8fae369766e9
    user:        mike@mike-notebook
    date:        Fri Nov 27 08:58:01 2009 +0300
```

```
summary:      Файл readme.txt добавлен в репозиторий
```

Итак, уже имеется ситуация, когда в локальном репозитории и в удаленном отличаются "головы" разработки, то есть существуют две различные ревизии, производные от одной. В терминах любой системы контроля версий — это ветвление, пусть пока неявное, но скоро станет таковым. Попробуем отправить имеющиеся ревизии в "центральный" репозиторий:

```
> hg outgoing
comparing with /home/mike/Repositories/newProject
searching for changes
changeset:   2:6872fa960507
tag:         tip
user:        mike@mike-vbox
date:        Sun Jan 10 19:40:45 2010 +0300
summary:         Файл second.txt создан во втором
репозитории

> hg push
pushing to /home/mike/Repositories/newProject
searching for changes
abort: push creates new remote heads!
(did you forget to merge? use push -f to force)
```

В данном примере Mercurial нам запрещает помещать изменения в центральные репозиторий, сообщая, что команда `push` приведет к созданию новой головы в удаленном репозитории. И предлагает произвести слияние репозиториев. Сделаем это:

```
> hg incoming
comparing with /home/mike/Repositories/newProject
searching for changes
changeset:   2:66c5686e355e
tag:         tip
user:        mike@mike-vbox
date:        Thu Jan 07 22:28:39 2010 +0300
summary:         Коммит файла first.txt в первом
репозитории

> hg pull
pulling from /home/mike/Repositories/newProject
searching for changes
adding changesets
adding manifests
adding file changes
```

```
    added 1 changesets with 1 changes to 1 files (+1
heads)
    (run 'hg heads' to see heads, 'hg merge' to merge)
```

Вытянув с "центрального" репозитория все имеющиеся изменения, Mercuial сообщает, что в локальном репозитории теперь две "головы" которые требуют слияния. Можно даже попросить Mercurial показать некоторую картинку (используется дополнение `graphlog`, расширение есть в стандартной поставке):

```
    > hg glog
    o  changeset:   3:66c5686e355e
    |  tag:         tip
    |  parent:      1:270e49e72f4b
    |  user:        mike@mike-vbox
    |  date:        Thu Jan 07 22:28:39 2010 +0300
    |   summary:       Коммит файла first.txt в первом
репозитории
    |
    | @  changeset:   2:6872fa960507
    |/   user:        mike@mike-vbox
    |    date:        Sun Jan 10 19:40:45 2010 +0300
    |     summary:      Файл second.txt создан во втором
репозитории
    |
    o  changeset:   1:270e49e72f4b
    |  user:        mike@mike-notebook
    |  date:        Fri Nov 27 10:39:35 2009 +0300
    |   summary:       Записан файл other.txt в другом
репозитории
    |
    o  changeset:   0:8fae369766e9
       user:        mike@mike-notebook
       date:        Fri Nov 27 08:58:01 2009 +0300
       summary:           Файл  readme.txt  добавлен  в
репозиторий
```

Поскольку пока не планировалось целенаправленно создавать две ветви разработки, необходимо выполнить слияние имеющихся ветвей:

```
    > hg merge
    1 files updated, 0 files merged, 0 files removed, 0
files unresolved
    (branch merge, don't forget to commit)
```

Итак, Mercurial, после команды `hg merge` произвел слияние рабочей копии и репозитория, и напоминает программисту, что эти изменения следовало бы подтвердить командой `commit`.

Теперь необходимо отправить изменения в "центральный" репозиторий, и посмотреть, что же делать теперь с ними первому разработчику.

```
> hg push
pushing to /home/mike/Repositories/newProject
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 1 files
```

Теперь переместимся в каталог первого разработчика, и получим изменения и из центрального репозитория:

```
> hg pull
pulling from /home/mike/Repositories/newProject
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 2 files
(run 'hg update' to get a working copy)
> hg update
1 files updated, 0 files merged, 0 files removed, 0
files unresolved
```

В трех репозиториях получена идентичная ситуация, несмотря на несколько более сложную исходную.

2.4 Основы организации ветвей

Для начала следует дать определения ветви, чтобы начинать с единого понимания процесса. Ветвь (branch) — это связанная последовательность ревизий (changeset) являющаяся отдельным направлением разработки. Таким образом, ветвь — это в первую очередь логическое понятие, так как в случае с распределенными системами контроля версий она будет содержать значительное число "спонтанных" ветвлений-слияний.

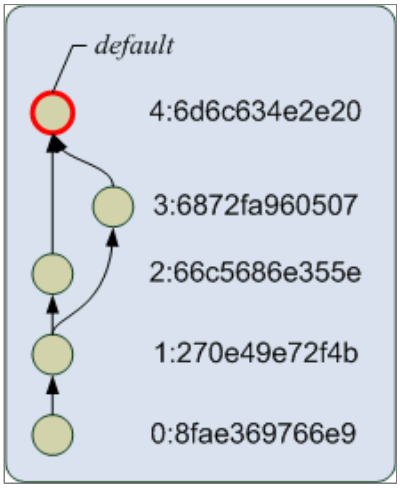

Рисунок 2.4 — Исходное состояние репозитория

В этом разделе продолжается работа над примером, описанном ранее. На рисунке 2.4 показано состояние репозитория после операций совершенных с ним в предыдущих разделах. И хотя формально в репозитории уже имеется одно ветвление, Mercurial говорит что ветвь одна:

```
> hg branches
default                        4:6d6c634e2e20
```

Команда `hg branches` выводит список всех именованных ветвей в репозитории. Как видно основная ветвь разработки называется default. Если быть точным, так называется ветвь, в которую происходит первое подтверждение изменений в репозиторий, так сказать название по умолчанию. На рисунке 2.4 приведено текущее состояние репозитория и граф ревизий в нем находящихся. Красным кружком отмечена "вершину" (`tip`) репозитория, как сказано в документации Mecurial, вершина — это самая свежая ревизия в репозитории. Команда `hg branches` не выводит анонимные ветви, хотя разработчики могут их использовать при необходимости, и создавать самостоятельно.

В Mercurial предусмотрен способ создания ветвей разработки с некоторыми именами, задаваемыми пользователем. Для организации подобных ветвлений предназначена команда `hg branch`. С помощью этой команды версия, находящаяся в локальной копии помечается ветвью с новым именем, при этом сама ветвь будет создана только после того, как будет выполнена команда `commit`. Далее реализуем именованную ветвь, родительской ревизией для которой будет ff8f:

```
> hg branch new_feature
marked working directory as branch new_feature
> hg commit
> hg branches
new_feature                    6:4d530267d302
```

```
default                                5:ff8ffd5270cb
```

Итак, как видно из результатов приведенных выше, Mercurial уже знает про две именованные ветви, "вершинами" для которых являются ревизии ff8f и 4d53, хотя на графе ревизий это одна ветвь. На рисунке 2.5 показано, что именно понимается под именованной ветвью в Mercurial, при этом, фактически, для каждой ветви есть своя "вершина" (`tip`), хотя `hg log` это не показывает.

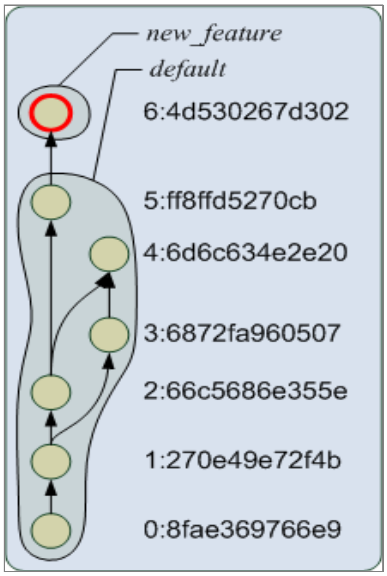

Рисунок 2.5 — Именованные ветви в репозитории

Убедиться в том, что "вершины" все таки существуют можно с помощью `hg update`, то есть переключившись на другую ветвь:

```
> hg update default
0 files updated, 0 files merged, 0 files removed, 0
files unresolved
> hg ident
ff8ffd5270cb
```

А затем переключится обратно:

```
> hg update new_feature
0 files updated, 0 files merged, 0 files removed, 0
files unresolved
$ hg ident
4d530267d302 (new_feature) tip
```

При этом в репозитории сложилась интересная ситуация. tip ветви default не совпадает с "головой" (head) этой же ветви. В этом легко убедиться, попросив Mercurial сказать какие же "головы" есть в репозитории:

```
    > hg heads
    changeset:   6:4d530267d302
    branch:      new_feature
    tag:         tip
    user:        mike@mike-vbox
    date:        Sun Jan 31 21:31:07 2010 +0300
    summary:           Создание именованной ветви в
репозитории

    changeset:   4:6d6c634e2e20
    parent:      3:6872fa960507
    parent:      2:66c5686e355e
    user:        mike@mike-vbox
    date:        Sun Jan 10 20:34:21 2010 +0300
    summary:     Выполнено слияние двух веток
```

Следует отметить один очень важный факт — ветвление произведено в локальном репозитории, и разработчик может работать с ним так, как ему угодно, при этом боясь поломать чужой код своими изменениями, или вызвать у коллег проблемы своими ветвлениями. Вот именно так концепция распределенной системы контроля версий позволяет решить стандартные болячки централизованных систем.

3. Порядок выполнения работы

3.1. Разработать модель командной работы согласно варианту, полученному у преподавателя.

3.2. Создать необходимое количество репозиториев, разработать соглашение по предназначению репозиториев.

3.3. Создать изменения в одном локальном репозитории, сохранить их в удаленном.

3.4. Получить набор изменений из удаленного репозитория в репозиторий отличный от описанного в п.3.3, внести дополнительные изменения и сохранить их в удаленном репозитории.

3.5. Внести одновременно разные изменения в локальные репозитории сохранить их все в удаленном, продемонстрировать процесс слияния.

3.6 Внести изменения в разных локальных репозиториях в одинаковые файлы, в одинаковых строках. Продемонстрировать процесс слияния при наличии конфликтов.

3.7. Продемонстрировать создание именованных веток в локальном репозитории.

3.8 Дать краткое описание команд распределенной системы контроля версий используемых при выполнении работы.

3.8. Проанализировать результаты работы, сделать выводы.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи, описание реализуемой модели работы команды разработчиков.

4.3. Команды, реализующие поставленную задачу, и результаты их работы.

4.4. Описание изменений в локальном и удаленном репозиториях на различных этапах работы.

4.5. Выводы по работе.

5. Контрольные вопросы

5.1. Расскажите о назначении систем контроля версий.

5.2. Опишите основные различия между централизованными и распределенными системами контроля версий.

5.3. Поясните понятие ревизии.

5.4. Назовите основные команды для работы с локальным репозиторием в Mercurial.

5.5. Опишите значение пометок присваиваемых файлам в результате выполнения команды `hg status` (`?`, `A`, `M`, `R`).

5.5. Опишите алгоритм и команды его реализующие для безопасного обмена ревизиями с удаленным репозиторием.

5.6. Поясните процесс возникновения конфликтов при слиянии репозиториев.

5.7. Расскажите об основных командах, используемых при разрешении конфликтов слияния, к какому результату они приводят?

5.6. Объясните понятие ветви в распределенной системе контроля версий.

5.7. Какие основные команды для работы с ветвями есть в Mercurial?

Лабораторная работа №2

Исследование распределенных системы контроля версий Git при коллективной разработке программных продуктов

1. Цель работы

Исследовать основные подходы к организации взаимодействия команды разработчиков с использованием распределенной системы контроля версий (DVCS). Приобрести практические навыки установки и настойки DVCS Git, организации ветвей разработки и осуществление слияния.

2. Основные положения

2.1 Создание Git-репозитория

Для создания Git-репозитория существуют два основных подхода. Первый подход — импорт в Git уже существующего проекта или каталога. Второй — клонирование уже существующего репозитория с сервера.

Создание репозитория в существующем каталоге

Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в проектный каталог и в командной строке ввести

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем `.git` содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем.

При необходимости добавления под версионный контроль существующих файлов (в отличие от пустого каталога), необходимо проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд `git add` указывающих индексируемые файлы, а затем `commit`:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

Теперь существует Git-репозиторий с добавленными файлами и начальным коммитом.

Клонирование существующего репозитория.

При необходимости получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна

команда `git clone`. Git получает копию практически всех данных, что есть на сервере. Каждая версия каждого файла из истории проекта забирается (pulled) с сервера, когда вы выполняете `git clone`.

Клонирование репозитория осуществляется командой `git clone [url]`:

```
$ git clone git://github.com/schacon/grit.git
```

Эта команда создаёт каталог с именем `grit`, инициализирует в нём каталог `.git`, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новый каталог grit, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от grit, можно это указать в следующем параметре командной строки:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван `mygrit`.

В Git'e реализовано несколько транспортных протоколов, которые можно использовать.

2.2 Запись изменений в репозиторий

На данном этапе имеется Git-репозиторий и рабочая копия файлов для некоторого проекта. Необходимо далее делать некоторые изменения и фиксировать “снимки” состояния (`snapshots`) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое требует сохранения.

Каждый файл в рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем слепке состояния проекта (`snapshot`); они могут быть неизменёнными, изменёнными или подготовленными к коммиту (`staged`). Неотслеживаемые файлы — это всё остальное, любые файлы в рабочем каталоге, которые не входили в последний слепок состояния и не подготовлены к коммиту. Когда впервые клонируется репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что их только взяли из хранилища (checked them out) и ничего пока не редактировалось.

Как только файлы будут отредактированы, Git будет рассматривать их как изменённые, т.к. они изменились их с момента последнего коммита. Необходим процесс индексации (`stage`) этих изменения и затем фиксация всех индексированные изменения, а затем цикл повторяется.

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`. Если выполнить эту команду сразу после клонирования, получится:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Это означает, что у есть чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. И наконец, команда сообщает на какой ветке (branch) сейчас находится система. Пока что это всегда ветка master — это ветка по умолчанию. Далее будет подробно рассказано про ветки и ссылки.

Предположим, добавлен в проект новый файл, простой файл README. Если этого файла раньше не было, и выполнить `git status`, получится неотслеживаемый файл:

```
$ vim README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#   README
nothing added to commit but untracked files present (use "git add" to track)
```

Понять, что новый файл README неотслеживаемый можно по тому, что он находится в секции "`Untracked files`" в выводе команды `status`. Статус "неотслеживаемый файл", по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока его явно об этом не попросить. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые не надо добавлять. Требовалось добавить README, сделаем это далее.

2.3 Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

```
$ git add README
```

Если снова выполнить команду `status`, то можно увидеть, что файл README теперь отслеживаемый и индексированный:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#   new file:   README
```

Видно, что файл проиндексирован по тому, что он находится в секции “Changes to be committed”. Если выполнить коммит в этот момент, то версия файла, существовавшая на момент выполнения команды `git add`, будет добавлена в историю снимков состояния. Команда `git add` принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет (индексирует) все файлы в данном каталоге.

2.4 Индексация изменённых файлов

Далее модифицируем файл, уже находящийся под версионным контролем. Если изменить отслеживаемый файл benchmarks.rb и после этого снова выполнить команду `status`, то результат будет следующим:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#   new file:   README
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be
committed)
#   modified:   benchmarks.rb
```

Файл benchmarks.rb находится в секции “Changes not staged for commit” — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду `git add` (это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния). Выполняется `git add`, чтобы проиндексировать benchmarks.rb, а затем снова выполняется `git status`:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
#   new file:   README
#   modified:   benchmarks.rb
```

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент, предположим, необходимо одно небольшое изменение в benchmarks.rb до фиксации. Необходимо открыть файл, внести и сохранить необходимые изменения и вроде бы готовы к коммиту. Ещё раз выполним `git status`:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#   new file:   README
#   modified:   benchmarks.rb
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be
committed)
#   modified:   benchmarks.rb
```

Теперь benchmarks.rb отображается как проиндексированный и непроиндексированный одновременно. Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда выполнялась команду `git add`. Если выполнить коммит сейчас, то файл benchmarks.rb попадёт в коммит в том состоянии, в котором он находился, на момент последнего выполнения команды `git add`, а не в том, в котором он находится в рабочем каталоге в момент выполнения `git commit`. Если изменился файл после выполнения `git add`, требуется снова выполнить `git add`, чтобы проиндексировать последнюю версию файла:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#   new file:   README
#   modified:   benchmarks.rb
```

2.5 Удаление файлов

Для того чтобы удалить файл из Git'a, необходимо удалить его из отслеживаемых файлов (точнее, удалить его из индекса) а затем выполнить коммит. Это позволяет сделать команда `git rm`, которая также удаляет

файл из рабочего каталога, так что в следующий раз не увидите его как “неотслеживаемый”.

Если просто удалить файл из рабочего каталога, он будет показан в секции “Changes not staged for commit” (“Изменённые но не обновлённые” — читай не проиндексированные) вывода команды git status:

```
$ rm grit.gemspec
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#       deleted:    grit.gemspec
```

Затем, если выполнить команду git rm, удаление файла попадёт в индекс:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#       deleted:    grit.gemspec
```

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если изменился файл и уже проиндексировали его, надо использовать принудительное удаление с помощью параметра -f. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git'a.

2.6 Перемещение файлов

В отличие от многих других систем версионного контроля, Git не отслеживает перемещение файлов явно. Когда переименовывается файл в Git'e, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — рассмотрим обнаружение перемещения файлов далее.

Наличие в Git'e команды mv выглядит несколько странным. Если необходимо переименовать файл в Git'e, можно сделать так:

```
$ git mv file_from file_to
```

На самом деле, если выполнить что-то вроде этого и посмотреть на статус, видно, что Git считает, что произошло переименование файла:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#       renamed:    README.txt -> README
```

Однако, это эквивалентно выполнению следующих команд:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

Git неявно определяет, что произошло переименование, поэтому неважно, переименование файла или использована команда `mv`. Единственное отличие состоит лишь в том, что `mv` — это одна команда вместо трёх — это функция для удобства. Важнее другое —можно использовать любой удобный способ, чтобы переименовать файл, и затем воспользоваться `add/rm` перед коммитом.

Другое полезное свойство— это удаление файл из индекса, одновременно оставив его при этом в рабочем каталоге. Другими словами, можно оставить файл, и убрать его из-под крнтроля Git'a. Это особенно полезно, если забыли добавить что-то в файл `.gitignore` и по ошибке проиндексировали, например, большой файл с логами, или промежуточные файлы компиляции. Чтобы сделать это, можно использовать опцию `--cached`:

```
$ git rm --cached readme.txt
```

В команду `git rm` можно передавать файлы, каталоги или `glob`-шаблоны. Это означает, что можно сделать так:

```
$ git rm log/\*.log
```

Следует обратить внимание на обратный слэш (\) перед *. Он необходим так как, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику командного интерпретатора. Эта команда удаляет все файлы, которые имеют расширение .log в каталоге log/. Или же можно сделать так:

```
$ git rm \*~
```

Эта команда удаляет все файлы, чьи имена заканчиваются на ~.

2.7 Просмотр истории коммитов

После того как созданы несколько коммитов, или же склонированы репозиторий с уже существующей историей коммитов, вероятно, появится необходимость просмотреть, что происходило с этим репозиторием. Наиболее простой и в то же время мощный инструмент для этого — команда `git log`.

Данные примеры используют очень простой проект, названный `simplegit`. Чтобы получить этот проект, следует выполнть:

```
git clone git://github.com/schacon/simplegit-progit.git
```

В результате выполнения `git log` в данном проекте, должен быть следующий результат:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
changed the version number
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
removed unnecessary test code
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
first commit
```

По умолчанию, без аргументов, `git log` выводит список коммитов созданных в данном репозитории в обратном хронологическом порядке. То есть самые последние коммиты показываются первыми. Эта команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем и электронной почтой автора, датой создания и комментарием.

Существует множество параметров команды `git log` и их комбинаций, для того чтобы показать именно то, что необходимо.

2.8 Отмена изменений

На любой стадии может возникнуть необходимость что-либо отменить. Рассмотрим несколько основных инструментов для отмены произведённых изменений. Следует быть осторожным, так как не всегда можно отменить

сами отмены. Это одно из немногих мест в Git'e, где можно потерять свою работу если сделаете что-то неправильно.

Изменение последнего коммита

Одна из типичных отмен происходит тогда, когда делается коммит слишком рано, забыв добавить какие-то файлы, или напуталы комментарии к коммиту. Если необходимо сделать этот коммит ещё раз, можно выполнить `commit` с опцией `--amend`:

```
$ git commit –amend
```

Эта команда берёт индекс и использует его для коммита. Если после последнего коммита не было никаких изменений (например, было запущена приведённая команда сразу после предыдущего коммита), то состояние проекта будет абсолютно таким же и всё, что изменится, это комментарий к коммиту. Появится всё тот же редактор для комментариев к коммитам, но уже с введённым комментарием к последнему коммиту. Можно отредактировать это сообщение, и оно перепишет предыдущее. Для примера, если после совершения коммита осознали, что забыли проиндексировать изменения в файле, которые хотели добавить в этот коммит, можно сделать так:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit –amend
```

Все три команды вместе дают один коммит — второй коммит заменяет результат первого.

Отмена индексации файла

В следующих двух разделах будет продемонстрировано, как переделать изменения в индексе и в рабочем каталоге. Команда, используемая для определения состояния этих двух действий, дополнительно напоминает о том, как отменить изменения в них. Приведём пример. Допустим, внесены изменения в два файла и необходимо записать их как два отдельных коммита, но случайно выполнен `git add *` и проиндексировали оба файла. Как теперь отменить индексацию одного из двух файлов? Команда `git status` напомнит об этом:

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#       modified:   README.txt
```

```
#       modified:   benchmarks.rb
```

Сразу после надписи “Changes to be committed”, написано использовать `git reset HEAD <файл>...` для исключения из индекса. Именно так следует отменять индексацию файла benchmarks.rb:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#       modified:   README.txt
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be
committed)
#     (use "git checkout -- <file>..." to discard
changes in working directory)
#       modified:   benchmarks.rb
```

Эта команда немного странновата, но она работает. Файл benchmarks.rb изменён, но снова не в индексе.

Отмена изменений файла

При условии, что нет необходимости оставлять изменения, внесённые в файл benchmarks.rb? Как быстро отменить изменения, вернуть то состояние, в котором он находился во время последнего коммита (или первоначального клонирования, или какого-то другого действия, после которого файл попал в рабочий каталог)? Команда `git status` говорит, как добиться и этого. В выводе для последнего примера, неиндексированная область выглядит следующим образом:

```
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be
committed)
#     (use "git checkout -- <file>..." to discard
changes in working directory)
#       modified:   benchmarks.rb
```

Из листинга видно как отменить сделанные изменения (в версии Git'a, начиная с 1.6.1). Следует выполнить следующее:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#       modified:   README.txt
```

Как видно, изменения были отменены. Следует понимать, что это опасная команда: все сделанные вами изменения в этом файле пропали — поверх файла просто скопирован другой файл. Не следует использовать эту команду, если нет полной уверенности, что этот файл вам не нужен. Если вам нужно просто сделать, чтобы файл не мешал, следует рассмотреть прятание (`stash`) и ветвление; эти способы обычно более предпочтительны. Всё что является частью коммита в Git'e, почти всегда может быть восстановлено. Даже коммиты, которые находятся на ветках, которые были удалены, и коммиты переписанные с помощью `--amend` могут быть восстановлены. Несмотря на это, всё, что никогда не попадало в коммит, скорее всего уже не увидеть снова.

3 Типовой сценарий работы с GIT системой

3.1 Начало работы

Рассмотрим работу системы контроля версий Git на примере конкретного проекта. Как и в любой другой системе контроля версий, все начинается с создания репозитория. Работа осуществляется с помощью команды `git`.

```
> git init
```

Результатом выполнения этой команды будет создание каталога `git`, который является базой, хранящей всю информацию о текущем репозитории.

Далее необходимо создать файл hello.html и зафиксировать его:

```
> touch hello.html
> git add hello.html
> git commit -m "First Commit"
```

Здесь, флаг `-m` означает, что далее последует комментарий к коммиту, если его не указать, то запустится редактор по умолчанию и предложит ввести его. Результатом будет следующий ответ системы:

```
[master (root-commit) 911e8c9] First Commit
 1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 hello.html
```

Из сообщения видно, что Git запомнил информацию о изменениях.

Так же, Git позволяет проверять состояние репозитория

```
> git status
# On branch master
nothing to commit (working directory clean)
```

Из данного сообщения видно, что в данный момент в репозитории нет изменений, которые нужно зафиксировать. Далее следует отредактировать файл hello.html и снова проверить статус. Результат представлен ниже:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#     (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   hello.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Из листинга видно, что Git обнаружил изменения в файле и даже предложил варианты действий. Здесь `git add <file>` позволяет указать, что текущее обновление будет зафиксировано, а `git checkout` позволит отменить изменения.

```
> git add hello.html
> git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   hello.html
#
```

Теперь видно, что изменения проиндексированы и могут быть применены с помощью команды commit, а так же предложен вариант отката изменений с помощью команды `git reset`.

Для команды `git commit` существует флаг `-a`, который позволяет автоматически выполнить индексацию и не вызывать команду `add`, однако им стоит пользоваться только тогда, когда вы уверены, что все изменения касаются только одного коммита. Разделение процесса индексации и сохранения позволяет четко определить какое изменение касается какого коммита.

Для просмотра истории проекта, используется команда `git log`:

```
> git log
commit 911e8c91caeab8d30ad16d56746cbd6eef72dc4c
Author: Author Name <sample (at) sample.com>
Date:   Wed Dec 12 23:02:54 2014 -0500

    First Commit
```

Напротив слова `commit` указана хеш-сумма, это один из важнейших элементов системы.

Команда `log` поддерживает форматирование вывода, например:

```
> git log --pretty=oneline
911e8c91caeab8d30ad16d56746cbd6eef72dc4c First Commit
```

Ниже показано как сделать лог действительно удобным и информативным:

```
> git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short
* 911e8c9 2014-12-03 | First Commit [Author Name]
```

Посмотреть подробную информацию по форматированию можно в официальной документации. Так же в изучении истории изменений помогают графические утилиты gitx (Mac) и gitk (Crossplatform).

Очевидно, что записывать всякий раз такую громоздкую команду неудобно. Git позволяет создавать псевдонимы (синонимы).

```
> git config --global alias.hist 'log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short'
```

Теперь, после использования команды `git hist` и пользователь увидим тот же результат. Так же изменения могут быть применены вручную, в файле `.gitconfig` в каталоге $home (cd ~/ для пользователей linux-based систем).

Выглядеть он будет примерно следующим образом:

```
[alias]
  hist = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph —date=short
```

Теперь требуется сохранить последнее изменение:

```
> git commit hello.html -m "Second Commit"
> git hist
* 43628f7 2011-03-09 | Added h1 tag [Author Name]
* 911e8c9 2011-03-09 | First Commit [Author Name]
```

В данном случае первый столбец, хранящий хеш, такой короткий по следующей причине: первые 7 символов являются идентифицирующими и их достаточно для работы.

```
>git checkout  911e8c9
Note: checking out '911e8c9'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 911e8c9... First Commit

>cat hello.html

«тут будет первоначальное содержимое файла»
```

В описании указан братно.о, что мы находимся в «мнимой» ветви HEAD. Требуется перейти о

```
> git checkout master

Previous HEAD position was 911e8c9... First Commit
Switched to branch 'master'
```

Удобной функцией является тегирование, которая позволяет создавать отметки.

```
> git tag v1
```

Эта команда закрепляет текущее состояние и теперь с помощью команды `checkout` есть возможность обращаться к ней не по хешу. Так же существует возможность указать `v1^` или `v1~1`, что будет означать возврат к версии предшествующей `v1`.

Команда `git tag` выведет все используемые теги.

3.2 Применение отмены изменений

Допустим вы изменили что-то в hello.html, но еще не проиндексировали, тогда

```
> git checkout hello.html
```

Эта команда вернет файл к последнему сохраненному состоянию.

Если уже выполнена индексация (`git add`), то воспользуемся командой reset.

```
> git reset HEAD hello.html
```

Эта команда сбросит буферную зону (смотрим основы git в первом разделе), однако, сам файл все еще содержит непроиндексированные изменения, тут делаем то же что и ранее

```
> git checkout hello.html
```

Если пользователь системы уже успел закоммитить изменения, то требуется использовать команду `git revert`

```
> git revert HEAD
```

После чего откроется редактор, в котором можно редактировать коммит-сообщение, так же можно добавить флаг `—no-edit`, который позволит проигнорировать внесение изменений.

После этого следует проверить лог, в нем будет указан и нежелательный коммит и коммит с откатом изменений.

Однако, часто не стоит загромождать историю откатами и бессмысленной информацией, для этого сохраним, на случай необходимости возврата текущее состояние

```
>git tag bad
```

Далее демонстрируется сброс коммитов:

```
> git reset —hard v1
```

(вместо `v1` можно использовать хеш)

После чего в истории изменений пользователь не увидит нежелательных коммитов.

Здесь флаг –hard указывает что каталог должен быть обновлен в соответствии с изменениями HEAD.

Однако, если в истории добавить флаг –all, то в таком случае снова будут видны те удаленные коммиты. Уничтожить их полностью можно с помощью сборщика мусора.

```
> git tag -d bad
```

После чего они будут удалены.

Бывают ситуации, когда пользователь уже совершили коммит, но забыли в нем что-то добавить, Git позволяет внести изменения в коммиты. Для этого внесите необходимые изменения в файлы, проиндексируйте их (add) и сделайте commit.

```
> git commit –amend -m «comment»
```

тут флаг amend и говорит, что текущий коммит — обновление предыдущего.

3.3 Работа с ветвями.

Работа с ветвями проекта является ключевым понятием при разработке сложных проектов командой разработчиков. Ветви используются для переключения контекстов, приостановки работы и выявления рисков. Некоторые пользователи создают ветвь "раздела" для каждой выполняемой задачи. Если они удовлетворены работой, они выполняют слияние с основной ветвью. Чтобы создать ветвь, используется команда checkout

```
>git checkout -b style
```

git checkout -b <имя_ветки> является краткой записью для команды git branch <имя_ветки> за которой идет git checkout <имя_ветки>.

В ходе работы пользователю потребовалось создать новый файл и сделать в нем несколько коммитов, после чего переключиться в ветку master и сделать собственный коммит. В результате, команда git hist -all будет использовать удобную псевдографику, например:

```
    * 6c0f848 2011-03-09 | Added README (HEAD, master)
[Sample Author]
    | * 07a2a46 2011-03-09 | Updated index.html (style)
[Sample Author]
```

```
    |  *  649d26c  2011-03-09  |  Hello  uses  style.css
[Sample Author]
    |  *  1f3cbd2  2011-03-09  |  Added  css  stylesheet
[Sample Author]
    |/
    *  8029c07  2011-03-09  |  Added  index.html.  [Sample
Author]
    *  567948a  2011-03-09  |  Moved  hello.html  to  lib
[Sample Author]
```

Слияние ветвей происходит с помощью команды `merge`. Перейдем в побочную ветвь и осуществим слияние:

```
> git merge master
```

В результате, появится сообщение похожее на приведенное ниже.

```
*   5813a3f 2011-03-09 | Merge branch 'master' into
style (HEAD, style) [Sample Author]
|\
|  *  6c0f848  2011-03-09  |  Added  README  (master)
[Sample Author]
* | 07a2a46 2011-03-09 | Updated index.html [Sample
Author]
*  |  649d26c  2011-03-09  |  Hello  uses  style.css
[Sample Author]
*  |  1f3cbd2  2011-03-09  |  Added  css  stylesheet
[Sample Author]
|/
*  8029c07  2011-03-09  |  Added  index.html.  [Sample
Author]
```

Однако, в ходе слияния могут возникать конфликтные ситуации. Далее рассмотрим пример возникновения конфликта и его разрешения. Для моделирования такой ситуации требуется вернуться в `master`, внести изменения и сделать коммит.

```
* 454ec68 2011-03-09 | Life is great! (HEAD, master)
[Sample Author]
| * 5813a3f 2011-03-09 | Merge branch 'master' into
style (style) [Sample Author]
| |\
| |/
|/|
*  |  6c0f848  2011-03-09  |  Added  README  [Sample
Author]
```

```
| * 07a2a46 2011-03-09 | Updated index.html [Sample Author]
| * 649d26c 2011-03-09 | Hello uses style.css [Sample Author]
```

После коммита «Added README» ветка master была объединена с веткой `style`, но в настоящее время в `master` есть дополнительный коммит, который не был слит с `style`.

```
> git checkout style
> git merge master
Auto-merging lib/hello.html
CONFLICT (content): Merge conflict in lib/hello.html
Automatic merge failed; fix conflicts and then commit the result.
```

Откройте конфликтующий файл, указанные в выводе команды и увидите что-то похожее на:

```
<<<<<<< HEAD
    текущая версия в ветке style
=======
    версия в master
>>>>>>> master
```

Решение конфликта производится вручную, после чего индексируется (add) и сохраняется(commit).

Так же есть альтернатива указанному варианту: `merge — rebase`.

3.4 Работа с удаленными репозиториями

Работа с удаленным репозиторем является основой командной разработки проектов. Для работы с удаленным репозиторием используем командой для создания копии удаленного репозитория — `git clone`:

```
> git clone hello cloned_hello
```

Перейдем в `cloned_hello` и посмотрим историю. В коммитах будет указана новая ветвь.

```
Updated index.html (HEAD, origin/master, origin/style, origin/HEAD, master)
```

`Origin` — указывает на то, что коммиты и ветви относятся к удаленному репозиторию.

Чтобы узнать имена удаленных репозиториев используется `git remote`.

Если добавить `git remote show <имя удаленного репозитория>` можно посмотреть подробную информацию о нем.

Если используется команда `git branch`, которая показывает список ветвей, то в данном случае увидите только `master`, однако, при добавлении флага `-a` можно увидеть все ветви, включая удаленные.

Если внесены изменения в удаленный репозиторий, их можно получить командой `git fetch`, которая получит новые коммиты, но не сольет их с вашими наработками в локальной ветке. Для слияния так же используем `git merge`, в данный момент с веткой `origin/master`.

Объединить эти 2 команды можно в одну:

```
> git pull,
```

что будет эквивалентно

```
>git fetch
>git merge origin/master
```

Для отправки изменений изначально работаем по стандартной схеме, индексируем изменения (add) и сохраняем их(commit), после чего используем команду push

```
>git push <куда> <откуда>
```

В `<откуда>` указывается ветвь, `master` по умолчанию.

В случае, если происходит работа с несколькими репозиториями, то в куда указывается название удаленного репозитория, которое можно увидеть в `git remote`.

3. Порядок выполнения работы

3.1. Разработать модель командной работы согласно варианту, полученному у преподавателя.

3.2. Создать необходимое количество репозиториев, разработать соглашение по предназначению репозиториев.

3.3. Создать изменения в одном локальном репозитории, сохранить их в удаленном.

3.4. Получить набор изменений из удаленного репозитория в репозиторий отличный от описанного в п.3.3, внести дополнительные изменения и сохранить их в удаленном репозитории.

3.5. Внести одновременно разные изменения в локальные репозитории сохранить их все в удаленном, продемонстрировать процесс слияния.

3.6. Продемонстрировать создание именованных веток в локальном репозитории.

3.7. Проанализировать результаты работы, сделать выводы.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи, описание реализуемой модели работы команды разработчиков.

4.3. Команды, реализующие поставленную задачу, и результаты их работы.

4.4.Описание изменений в локальном и удаленном репозиториях на различных этапах работы.

4.5. Выводы по работе.

5. Контрольные вопросы

5.1. Расскажите о назначении систем контроля версий.

5.2.Опишите основные различия между централизованными и распределенными системами контроля версий.

5.3. Поясните понятие ревизии.

5.4. Назовите основные команды для работы с локальным репозиторием в Git.

5.5. Опишите алгоритм и команды его реализующие для безопасного обмена ревизиями с удаленным репозиторием.

5.6. Объясните понятие ветви в распределенной системе контроля версий.

5.7. Какие основные команды для работы с ветвями есть в Git?

Библиографический список

1. Чакон С., Штрауб Б. Git для профессионального программиста / С. Чакон, Б. Штрауб — СПб.: Питер, 2016. — 496 с.: ил.

2. Основы работы с Git [Электронный ресурс]. – Режим доступа: http://www.calculate-linux.ru/main/ru/git. – Основы работы с Git. – (Дата обращения: 03.11.2015).

3. Основы Меркуриала: объединение работы t [Электронный ресурс]. – Режим доступа: http://dreamhelg.ru/2009/09/mercurial-merge-work/. – Основы Меркуриала: объединение работы. — (Дата обращения: 03.11.2015).

4. Mercurial.ru [Электронный ресурс]. – Режим доступа: http://www.mercurial.ru. – Mercurial.ru. — (Дата обращения: 03.11.2015).

5. O'Sullivan B. Mercurial: The Definitive Guide / — O'Reilly Media, Inc, 2009.

6. Loeliger J., McCullough M. Version Control with Git: Powerful tools and techniques for collaborative software development — 2nd ed. / — O'Reilly Media Inc, 2012.

7. Muller C. Instant Mercurial Distributed SCM Essentials How-to / — Packt Publishing Ltd, 2013.

8. Chacon S., Straub B. Pro Git 2nd ed./ .— Apress, 2014.