Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

УПРАВЛЕНИЕ ЖИЗНЕННЫМ ЦИКЛОМ ПРОГРАММНЫХ СИСТЕМ

Методические указания к практическим занятиям и организации самостоятельной работы для студентов направления «Программная инженерия»

(уровень бакалавриата)

2018

Масляев Владимир Сергеевич

Управление жизненным циклом программных систем: Методические указания к практическим занятиям и организации самостоятельной работы для студентов направления «Программная инженерия» (уровень бакалавриата) / В.С. Масляев. – Томск, 2018. – 51 с.

© Томский государственный университет систем управления и радиоэлектроники, 2018

© Масляев В.С., 2018

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ 4

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРОВЕДЕНИЮ ПРАКТИЧЕСКИХ ЗАНЯТИЙ 5

2.1 Практическое занятие «Первоначальная настройка git» 5

2.2 Практическое занятие «Игнорирование, сравнение, удаление и перемещение файлов» 13

2.3 Практическое занятие «Просмотр истории коммитов» 21

2.4 Практическое занятие «Отмена изменений. Работа с метками» 30

2.5 Практическое занятие «Ветвление. Конфликты» 37

3 МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОРГАНИЗАЦИИ САМОСТОЯТЕЛЬНОЙ РАБОТЫ 45

3.1 Общие положения 46

3.2 Проработка лекционного материала и подготовка к практическим занятиям 46

3.3 Контрольная работа «Обзор современной системы / методологии, помогающей управлять жизненным циклом» 47

3.4 Подготовка к экзамену 48

4 РЕКОМЕНДУЕМЫЕ ИСТОЧНИКИ 51

Введение

Данное учебно-методическое пособие предназначено для подготовки и выполнения практических занятий по дисциплине «Управление жизненным циклом программных систем» для студентов направления «Программная инженерия».

Практические занятия по данной части дисциплины имеют целью: закрепление теоретического материала, получение навыков самостоятельной работы с системой контроля версий Git.

Цель изучения дисциплины «Управление жизненным циклом программных систем» является формирование у студентов профессиональных знаний, умений и навыков о методах и средствах управления жизненным циклом информационных систем, использование информационных технологий на всех стадиях их жизненного цикла.

Задачи:

получение практических и теоретических навыков использования информационных технологий на всех этапах жизненного цикла информационных систем;

формирование умений решения задач хранения информации на различных этапах жизненного цикла;

получение опыта управления жизненным циклом программных систем;

приобретение навыков использования систем контроля версий в области управления жизненным циклом информационных систем;

изучение современных информационных технологий необходимых для управления проектами.

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРОВЕДЕНИЮ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

2.1 Практическое занятие «Первоначальная настройка git»

Цель работы: провести первоначальную настройку системы контроля версии git, после установки инициализировать каталог для работы, разобраться с существующими состояниями файлов в git, сделать первый коммит.

Установка имени

В состав git'а входит утилита **git config**, которая позволяет просматривать и устанавливать параметры, контролирующие все аспекты работы git'а и его внешний вид.

Первое, что необходимо сделать после установки git'а, — указать имя и адрес электронной почты. Это важно, потому что каждый коммит в git'е содержит эту информацию, и она включена в коммиты, передаваемые разработчиками, и не может быть далее изменена.

Для того чтобы задать имя используется команда:

```
$ git config --global user.name "My name"
```

Для задания e-mail'а используется следующая команда:

```
$ git config --global user.email my_name@example.com
```

Опция **--global** говорит о том, что эти настройки достаточно сделать только один раз, поскольку в этом случае git будет использовать эти данные для всего, что разработчик делает в этой системе. Если для каких-то отдельных проектов необходимо указать другое имя или электронную почту, можно выполнить эту же команду без параметра **--global** в каталоге с нужным проектом.

Выбор редактора

В git также можно выбрать текстовый редактор, который будет использоваться, если будет нужно набрать сообщение. По умолчанию git использует стандартный редактор системы, обычно это Vi или Vim. Если необходимо использовать другой текстовый редактор, можно сделать следующее:

```
$ git config --global core.editor emacs
```

Утилита сравнения

Для установки встроенной diff-утилиты, которая будет использоваться для разрешения конфликтов слияния, необходимо выполнить следующую команду:

$ git config --global merge.tool meld

Git умеет делать слияния при помощи различных утилит, таких как kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge и opendiff, но также возможна настройка и другой утилиты.

Проверка настроек

Для того чтобы проверить используемые настройки, необходимо использовать команду **git config --list**, чтобы показать все, которые git найдёт:

$ git config --list

Результат:

user.name=My name
user.email=my_name@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto

Также можно проверить значение конкретного ключа, выполнив **git config {ключ}**, например, проверим содержимое параметра user.name:

$ git config user.name

Результат:

Scott Chacon

Помощь в git

Если нужна помощь при использовании git'a, есть три способа открыть страницу руководства по любой команде git'a:

$ git help <команда>
$ git <команда> --help
$ man git-<команда>

Например, можно открыть руководство по команде config:

$ git help config

Эти команды хороши тем, что ими можно пользоваться всегда, даже без подключения к сети.

Создание репозитория в существующем каталоге

Для того чтобы начать использовать git для существующего проекта, 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 проект ещё не находится под версионным контролем. Данная команда инициализирует возможность работы с git, но не вносит файлы под контроль.

Запись изменений в репозиторий

Теперь, когда проинициализирован git-репозиторий необходимо производить работу с файлами, делать некоторые изменения и фиксировать "снимки" состояния (*snapshots*) этих изменений в репозитории каждый раз, когда проект достигает состояния, которое хотелось бы сохранить.

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 (*unmodified*), изменёнными (*modified*) или подготовленными к коммиту (*staged*). Неотслеживаемые файлы *(untracked)* — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту.

Как только происходит редактирование файлов, git будет рассматривать их как изменённые, т.к. они изменились с момента последнего коммита. Вы индексируете (*stage*) эти изменения и затем фиксируете все индексированные изменения, а затем цикл повторяется. Этот жизненный цикл изображён на рисунке 2.1.

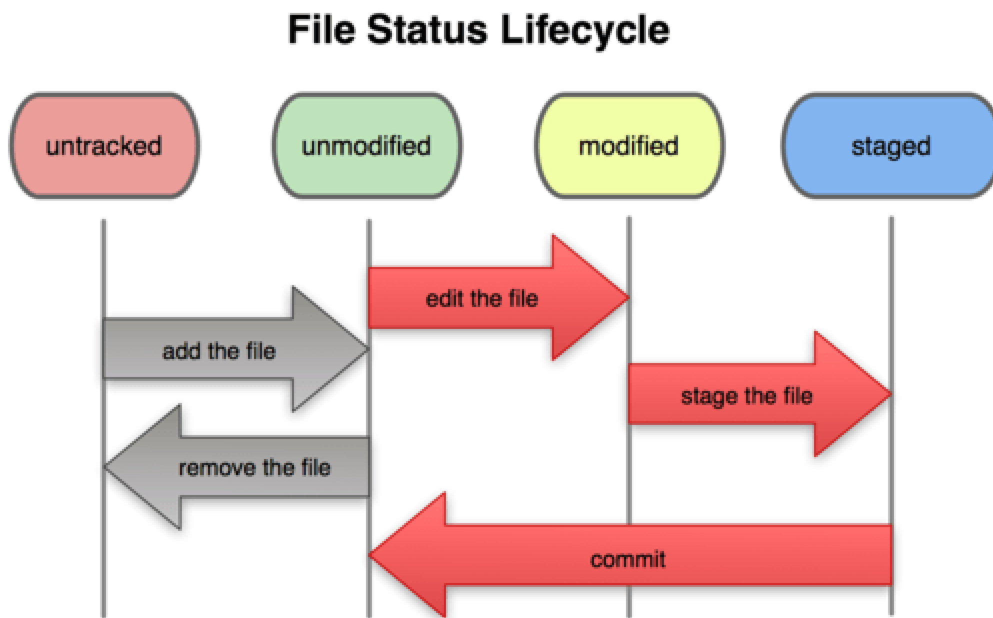

Рисунок 2.1 - Жизненный цикл состояний файлов.

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда **git status**. Если выполнить эту команду сразу после создания репозитория, получится результат подобного вида:

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

Это означает, что на данный момент в репозитории 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*) происходит работа. Сейчас это ветка *master* — это ветка по умолчанию.

Произведем добавление в проект нового файла, например простой файл README. После чего произведем выполнение команды **git status**, в результате появится неотслеживаемый файл:

```
$ 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 **git status**. Статус "неотслеживаемый файл", означает, что git видит файл, отсутствующий в предыдущем снимке состояния (коммите); git не станет добавлять его в коммиты, пока файл не будет переведен в секцию отслеживаемые. Это предостерегает от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые не должны попасть в проект.

Отслеживание новых файлов

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

```
$ git add README
```

Если теперь снова выполнить команду **git status**, то обнаружим, 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
```

Теперь файл проиндексирован потому, что он находится в секции “Changes to be committed”. Команда **git add** принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет (индексирует) все файлы в данном каталоге.

Фиксация изменений

Теперь, когда была произведена настройка индекса, зафиксируем изменения. Всё, что не проиндексировано — любые файлы, созданные или изменённые, и для которых не выполнена команда **git add** после момента редактирования — не войдут в коммит. Они останутся изменёнными файлами на диске. Добавим под версионный контроль созданный файл README. Простейший способ зафиксировать изменения — это выполнить команду **git commit**:

$ git commit

Эта команда откроет выбранный текстовый редактор. (Редактор устанавливается системной переменной *$EDITOR* — обычно это vim или emacs.

В редакторе будет отображён следующий текст:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Комментарий по умолчанию для коммита содержит закомментированный результат работы команды git status и ещё одну пустую строку сверху. Для ещё более подробного напоминания, что же именно было изменено, можно передать аргумент **-v** в команду **git commit**. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом можно точно увидеть всё, что сделано. Когда будет произведен выход из редактора, git создаёт коммит с этим сообщением (удаляя комментарии и вывод diff'a).

Для того чтобы ввести комментарий в текстовом редакторе vim необходимо:

- нажать клавишу I (insert), это переведет редактор в режим ввода данных;
- ввести свой комментарий к коммиту;
- нажать Esc, для выхода из режима ввода данных;
- написать :wq (write quit), что означает записать и выйти, после чего нажать Enter.

Указание комментария при коммите является обязательной процедурой, так как если комментарий не будет написан, то git выдаст сообщение об ошибке следующего вида:

```
# Aborting commit due to empty commit message.
```

Индексация измененных файлов

Модифицируем файл, уже находящийся под версионным контролем. Так как созданный файл README уже помещен под версионный контроль, изменим его и после этого снова произведем выполнение команды **git status**, результат будет примерно следующим:

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

Файл README 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). Выполним git add, чтобы проиндексировать README, а затем снова выполним **git status**:

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

Теперь файл проиндексирован и войдет в следующий коммит. Произведем еще небольшое изменение в файле README перед фиксацией. Откроем файл, внесем изменения и сохраним их. Теперь выполним еще раз команду **git status**:

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

```
#   modified:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   README
#
```

Теперь README 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 команда **git add**. Если выполнить коммит сейчас, то файл README попадёт в коммит в том состоянии, в котором он находился, когда последний раз выполнялась команда **git add**, а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения **git commit**. Если производилось изменение файла после выполнения **git add**, придётся снова выполнить **git add**, чтобы проиндексировать последнюю версию файла.

Порядок выполнения практической работы

1. Изучить теоретическую часть работы.
2. Зайти в папку T://{Номер группы} и в ней создать папку соответствующую инициалам студента на английском языке. Например, для студента Иванов Петр Петрович, папка будет иметь имя IPP.
3. Провести инициализацию репозитория в созданной папке. Для этого, открыть программу **Git Bash**, перейти в созданную папку (для перемещения используется команда **cd T://{Номер группы}/{Инициалы}**).
4. Установить настройки имени и e-mail’а, не используя опцию **--global.**
5. Создать в папке файл my_first_file.txt и проиндексировать его.
6. Сделать первый коммит.
7. Открыть файл my_first_file.txt и добавить в него строчку “test row”. Проиндексировать изменения.
8. Создать новый файл my_second_file.txt. Проиндексировать изменения.
9. Сделать второй коммит.
10. Продемонстрировать преподавателю ход работы, ответить на уточняющие вопросы.

2.2 Практическое занятие «Игнорирование, сравнение, удаление и перемещение файлов»

Цель работы: научиться исключать файлы, которые нет необходимости вести в системе контроля версий. Получить практические навыки сравнения проделанных изменений в файлах.

Игнорирование файлов

Зачастую, имеется группа файлов, которые не только нет необходимости автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т.п.). В таком случае, необходимо создать файл *.gitignore* с перечислением шаблонов соответствующих таким файлам. Приведем пример файла .gitignore, для этого откроем его с помощью утилиты *cat (*UNIX-утилита, выводящая последовательно указанные файлы (или устройства), таким образом, объединяя их в единый поток*)*:

```
$ cat .gitignore
*.[oa]
*~
```

Первая строка предписывает git'у игнорировать любые файлы заканчивающиеся на .o или .a — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например Emacs, для обозначения временных файлов. Также можно включить каталоги log, tmp или pid; автоматически создаваемую документацию; и т.п. Хорошая практика заключается в настройке файла .gitignore до того, как начать серьёзно работать, это защитит от случайного добавления в репозиторий файлов, которых там быть не должно.

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с # (символ комментария), игнорируются.
- Можно использовать стандартные glob шаблоны.
- Можно заканчивать шаблон символом слэша (/) для указания каталога.
- Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения используемые командными интерпретаторами. Символ * соответствует 0 или более

символам; последовательность [abc] — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; [0-9] соответствует любому символу из интервала (в данном случае от 0 до 9).

Вот ещё один пример файла .gitignore:

```
# комментарий — эта строка игнорируется
# не обрабатывать файлы, имя которых заканчивается на .a
*.a
# НО отслеживать файл lib.a, несмотря на то, что мы игнорируем все .a файлы с помощью предыдущего правила
!lib.a
# игнорировать только файл TODO находящийся в корневом каталоге, не относится к файлам вида subdir/TODO
/TODO
# игнорировать все файлы в каталоге build/
build/
# игнорировать doc/notes.txt, но не doc/server/arch.txt
doc/*.txt
# игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

Шаблон **/ доступен в Git, начиная с версии 1.8.2.

Создание файла .gitignore на Windows

Так как в современных версиях Windows, начиная с 7, нельзя создать файл с пустым названием, таким как .gitignore, существует несколько вариантов обхода данной ситуации.

Первый вариант, содать файл с именем .gitignore. при этом необходимо убедится, что в системе отключена опция скрывать расширения, иначе создаться файл .gitignore..txt.

Второй вариант, создать файл gitignore.txt и выполнить следующую команду в командной строке:

```
ren gitignore.txt .gitignore
```

Просмотр индексированных и неиндексированных изменений

Результат работы команды **git status** не всегда является достаточно информативен, иногда требуется знать, что конкретно поменялось, а не только какие

файлы были изменены — для просмотра данных изменений используется команда **git diff**. Обычно данную команду используют для получения ответов на два вопроса: что изменилось, но ещё не проиндексировано, и что проиндексировано будет зафиксировано. Если **git status** отвечает на эти вопросы слишком обобщённо, то **git diff** показывает непосредственно добавленные и удалённые строки — собственно заплатку (*patch*).

Внесем изменения и проиндексируем файл README, а затем изменим файл my_first_file.txt, который уже находится под версионным контролем, без индексирования. Если выполнить команду **git status**, получим следующий результат:

```
$ 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:   my_first_file.txt
#
```

Чтобы увидеть, что же было изменено, но пока не проиндексировано, наберем команду **git diff** без аргументов:

```
$ git diff
diff --git a/my_first_file.txt b/my_first_file.txt
index 3cb747f..da65585 100644
--- a/my_first_file.txt
+++ b/my_first_file.txt
@@ -36,6 +36,10 @@ def main
        @commit.parents[0].parents[0].parents[0]
end
+       run_code(x, 'commits 1') do
+         git.commits.size
+       end
+
```

```
run_code(x, 'commits 2') do
 log = git.commits('master', 15)
 log.size
```

Эта команда сравнивает содержимое рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если необходимо посмотреть, что проиндексировано и что войдёт в следующий коммит, необходимо выполнить **git diff --cached**. (В git'e версии 1.6.1 и выше, имеется аналог, который также можно использовать **git diff --staged**) Эта команда сравнивает индексированные изменения с последним коммитом:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+ some text
+ we add into file
+
+ good day
```

Важно отметить, что **git diff** сама по себе не показывает все изменения сделанные с последнего коммита — только те, что ещё не проиндексированы. Такое поведение может сбивать с толку, так как если проиндексировать все изменения, то **git diff** ничего не вернёт.

Другой пример: проиндексировали файл README и затем изменили его, для этого используем утилиту echo (Unix команда, предназначенная для отображения строки текста, может служить для записи строки в файл, если используется > файл будет перезаписан, если >> строка будет дописана в конец файла), можно использовать **git diff** для просмотра как индексированных изменений в этом файле, так и тех, что пока не проиндексированы:

```
$ git add README
$ echo '# test line' >> README
$ git status
# On branch master
#
```

```
# Changes to be committed:
#
#   modified:   README
#
# Changes not staged for commit:
#
#   modified:   README
#
```

Теперь используя команду **git diff** посмотрим непроиндексированные изменения:

```
$ git diff
diff --git a/README b/README
index e445e28..86b2f7c 100644
--- a/README
+++ b/README
@@ -127,3 +127,4 @@

good day
+# test line
```

А также уже проиндексированные, используя **git diff --cached**:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+ some text
+ we add into file
+
+ good day
```

Удаление файлов

Для того чтобы удалить файл из git'а, 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 файл не будет как “неотслеживаемый”.

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

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

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

```
$ git rm somefile.txt
rm 'somefile.txt'
$ 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 было произведено изменение файла и после он был проиндексирован, придется 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'а.

Другая полезная функция, которую полезно использовать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, необходимо

оставить файл на винчестере, и убрать его из-под версионного контроля git'a. Это особенно полезно, если что-то по ошибке не было добавлено в файл .gitignore и по ошибке проиндексировано, например, большой файл с логами, или множество промежуточных файлов компиляции. Чтобы сделать это, необходимо использовать опцию **--cached**:

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

В команду **git rm** можно передавать файлы, каталоги или glob-шаблоны. Приведем пример:

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

Обратим внимание на обратный слэш (\) перед *. Он необходим из-за того, 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/. Еще один пример:

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

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

Перемещение файлов

В отличие от многих других систем версионного контроля, git не отслеживает перемещение файлов явно. Когда происходит переименовывание файла в git'e, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован.

Таким образом, наличие в git'e команды **git mv** выглядит несколько странным. Если необходимо переименовать файл в git'e, можно сделать следующее:

```
$ git mv file_from file_to
```

А теперь просмотрим статус. 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, переименуете вы файл так или используя команду **git mv**. Единственное отличие состоит лишь в том, что **git mv** — это одна команда вместо трёх — это функция для удобства. Можно использовать любой удобный способ, чтобы переименовать файл, и затем воспользоваться add/rm перед коммитом.

Порядок выполнения практической работы

1. Изучить теоретическую часть работы.
2. Продолжить работу с созданным репозиторием на первой практической работе.
3. Создать папку temp и вложенную в нее папку bin в своем репозитории.
4. Создать папку log и добавить в нее 2 файла, main.html и some.tmp.
5. Создать файл .gitignore и добавить в игнорирование файлы или папки согласно варианту.
6. Закоммитить добавление файла .gitignore.
7. Внести изменения в файл my_first_file.txt добавив строчку “row to index”, проиндексировать данные изменения. Еще раз внести изменения в файл, добавив строчку “row no index”.
8. Посмотреть индексированные и неиндексированные изменения используя команду **git diff**.
9. Удалить файл my_first_file.txt, зафиксировать данное удаление.
10. Переименовать файл my_second_file.txt в my_first_file.txt, зафиксировать изменение.
11. Продемонстрировать преподавателю ход работы, ответить на уточняющие вопросы.

Варианты

Номер	Задание
1	Папку temp и файлы с расширением .tmp из папки log.
2	Папку temp, кроме файлов в ней с расширением .doc.
3	Папку temp и log, кроме файлов в log с расширением .html.
4	Папку temp, исключая вложенную папку bin.
5	Папку log, и вложенную в папку temp папку bin

2.3 Практическое занятие «Просмотр истории коммитов»

Цель работы: освоить механизм работы с командой git log для получения информации об истории коммитов.

Создание коммита с комментарием

Существует возможность создать коммит совместно со своим комментарием сразу в командной строке, для этого используется команда **git commit** с параметром **-m**, приведем пример:

```
$ git commit -m "My comment for commit"
[master]: created 463dc4f: "My comment for commit"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Игнорирование индексации

Несмотря на то, что индекс является очень полезным для создания коммитов временами он может увеличивать время работы. Имеется механизм чтобы пропустить этап индексирования, git предоставляет простой способ. Добавление параметра **-a** в команду **git commit** заставляет git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя обойтись без **git add**:

```
$ git status
# On branch master
#
# Changes not staged for commit:
```

```
#
#   modified:   README
#
$ git commit -a -m 'added new README'
[master 83e38c7] added new README
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Автоматическое дополнение

Нажав Tab во время ввода команды для Git'a, получаем набор вариантов на выбор:

```
$ git co<tab><tab>
commit config
```

В данном случае, набрав git co и дважды нажав клавишу Tab, получаем как варианты commit и config. Добавление m<tab> выполнит дополнение до **git commit** автоматически.

То же самое работает и для опций. Например, если необходимо выполнить команду **git log** и не помните какую-либо опцию, начинаем её печатать и затем нажимаем Tab, чтобы увидеть, что подходит:

```
$ git log --s<tab>
--shortstat  --since=  --src-prefix=  --stat   --summary
```

Эта возможность позволяет сэкономить рабочее время от чтения документации и легко применяется.

Просмотр истории коммитов

После того как будет создано несколько коммитов, вероятнее всего появиться необходимость просмотреть, что же происходило с этим репозиторием. Наиболее простой и в то же время мощный инструмент для этого — команда **git log**.

В результате выполнения **git log** получиться подобный 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. Как видно из примера, эта команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем, электронной почтой автора, датой создания и комментарием.

Существует большое количество параметров команды **git log** и их комбинаций, для того чтобы показать именно то, что необходимо найти. Рассмотрим наиболее часто применяемые.

Один из наиболее полезных параметров — это **-p**, который показывает дельту (разницу/diff), привнесенную каждым коммитом. Также можно использовать **-2**, что ограничит вывод до 2-х последних записей:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      =   "simplegit"
```

```
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

Этот параметр показывает ту же самую информацию плюс внесённые изменения, отображаемые непосредственно после каждого коммита. Это очень удобно для инспекций кода или для того, чтобы быстро посмотреть, что происходило в результате последовательности коммитов, добавленных коллегой.

В некоторых ситуациях гораздо удобней просматривать внесённые изменения на уровне слов, а не на уровне строк. Чтобы получить дельту по словам вместо обычной дельты по строкам, нужно дописать после команды **git log -p** опцию **--word-diff**. Дельты на уровне слов практически бесполезны при работе над программным кодом, но они будут очень кстати при работе над длинным текстом, таким как книга или диссертация. Рассмотрим пример:

```
$ git log -U1 --word-diff
```

```
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

Как видно, в этом выводе нет ни добавленных, ни удаленных строк, как для обычного diff'а. Вместо этого изменения показаны внутри строки. Добавленное слово заключено в {+ +}, а удалённое в [- -]. Также может быть полезно сократить обычные три строки контекста в выводе команды diff до одной строки, так как контекстом в данном случае являются слова, а не строки. Сделать это можно с помощью опции **-U1** как было показано в примере выше.

Можно также просматривать лог коммита, используя его хеш сумму, допускается использование сокращенного значения, например:

```
git log 136556ff0bb8133763632a4558792fd931a597e0
```

С командой **git log** также можно использовать группы суммирующих параметров. Например, если необходимо получить некоторую краткую статистику по каждому коммиту, можно использовать параметр **--stat**:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 files changed, 0 insertions(+), 5 deletions(-)
```

Как видно из лога, параметр **--stat** выводит под каждым коммитом список изменённых файлов, количество изменённых файлов, а также количество добавленных и удалённых строк в этих файлах. Он также выводит сводную информацию в конце. Другой полезный параметр — это **--pretty**. Он позволяет изменить формат вывода лога. Существует несколько доступных предустановленных вариантов. Параметр **oneline** выводит каждый коммит в одну строку, что удобно если необходимо просмотреть большое количество коммитов. В дополнение к этому, параметры **short**, **full**, и **fuller**, практически не меняя формат вывода, позволяют выводить меньше или больше деталей соответственно:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Наиболее интересный параметр — это **format**, который позволяет полностью создать собственный формат вывода лога. Это особенно полезно, когда необходимо создать отчёты для автоматического разбора (парсинга) — поскольку явно задан формат и пользователь уверен в том, что он не будет изменяться при обновлениях git'a:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Основные опции доступные для параметра **format** представлены в Таблице 2.1.

Таблица 2.1 - Список параметров формата.

Параметр	Описание выводимых данных
%H	Хеш коммита
%h	Сокращённый хеш коммита
%T	Хеш дерева
%t	Сокращённый хеш дерева
%P	Хеши родительских коммитов
%p	Сокращённые хеши родительских коммитов
%an	Имя автора
%ae	Электронная почта автора
%ad	Дата автора (формат соответствует параметру --date=)
%ar	Дата автора, относительная (пр. "2 мес. назад")
%cn	Имя коммитера
%ce	Электронная почта коммитера
%cd	Дата коммитера
%cr	Дата коммитера, относительная
%s	Комментарий

Разница между *автором* и *коммитером*. Автор — это человек, изначально сделавший работу, тогда как коммитер — это человек, который последним применил эту работу. Так что если вы послали патч (заплатку) в проект и один из основных разработчиков применил этот патч, вы оба не будете забыты — вы как автор, а разработчик как коммитер.

Ограничение вывода команды git log

Кроме опций для форматирования вывода, **git log** имеет ряд полезных ограничительных параметров, то есть параметров, которые дают возможность отобразить часть коммитов. Один из таких параметров уже был рассмотрен —

параметр **-2**, который отображает только два последних коммита. На самом деле, можно задать **-<n>**, где **n** это количество отображаемых коммитов.

Существуют параметры, ограничивающие по времени, такие как **--since** и **--until**. Например, следующая команда выдаёт список коммитов, сделанных за последние две недели:

```
$ git log --since=2.weeks
```

Такая команда может работать с множеством форматов — можно указать точную дату (“2016-01-15”) или относительную дату, такую как “2 years 1 day 3 minutes ago”.

Также существует возможность отфильтровать список коммитов по какому-либо критерию поиска. Опция **--author** позволяет фильтровать по автору, опция **--grep** позволяет искать по ключевым словам в сообщении. (Заметим, что, если указана и опция **author**, и опция **grep**, то будут найдены все коммиты, которые удовлетворяют первому ИЛИ второму критерию. Чтобы найти коммиты, которые удовлетворяют первому И второму критерию, следует добавить опцию **--all-match**.)

Еще одна полезная опция-фильтр для **git log** — это путь. Указав имя каталога или файла, можно ограничить вывод **log** теми коммитами, которые вносят изменения в указанные файлы. Эта опция всегда указывается последней и обычно предваряется двумя минусами (**--**), чтобы отделить пути от остальных опций.

Приведем пример, необходимо посмотреть из истории git'a такие коммиты, которые вносят изменения в тестовые файлы, были сделаны Ivan Petrov, не являются слияниями и были сделаны в октябре 2015-го, выполним команду:

```
$ git log --pretty="%h - %s" --author="Ivan Petrov" --since="2015-10-01" \
  --before="2015-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Порядок выполнения практической работы

1. Изучить теоретическую часть работы.
2. Продолжить работу с созданным репозиторием.
3. Изучить возможности команды git log, выполнить различные варианты вывода информации и ее отбора.

4. Выполнить задание согласно варианту.

5. Продемонстрировать преподавателю ход работы, ответить на уточняющие вопросы.

Варианты

Номер	Задание
1	Вывести коммиты, автором которых являетесь Вы, за последний месяц.
2	Вывести все коммиты в формате: короткий хеш, автор, комментарий.
3	Вывести все коммиты в сообщении которых присутствует слово my.
4	Вывести все коммиты за февраль 2016 с информацией о том, какие файлы были изменены.
5	Вывести информацию о первом коммите в системе, с выводом дельты (diff).
6	Вывести коммиты сделанные за последний месяц назад, но исключая последнюю неделю.
7	Вывести информацию о коммитах в формате: автор, дата, а также со списком измененных файлов.
8	Вывести коммиты, автором которых являетесь Вы, с выводом дельты (diff).
9	Вывести коммиты, в которых происходили изменения файла my_first_file.txt, за последние 2 недели.
10	Вывести последние 3 коммита в формате: автор, комментарий.
11	Вывести информацию о первом коммите в формате: дата, автор, комментарий, а также список измененных файлов.
12	Вывести коммиты, автором которых являетесь Вы, со списком измененных файлов.
13	Вывести все коммиты за февраль 2016 в формате: сокращенный хеш, дата, комментарий.

14	Вывести все коммиты в формате: e-mail автора, дата коммита, хеши родительских коммитов.
15	Вывести коммиты, в которых происходили изменения файла my_first_file.txt.

2.4 Практическое занятие «Отмена изменений. Работа с метками»

Цель работы: научиться отменять сделанные изменения, работать с метками.

На любой стадии может возникнуть необходимость что-либо отменить. Рассмотрим несколько основных инструментов для отмены произведённых изменений. Необходимо помнить что не всегда можно отменить сами отмены. Это одно из немногих мест в git'е, где можно потерять выполненную работу если сделаете что-то неправильно.

Изменение последнего коммита

Одна из типичных отмен происходит тогда, когда коммит сделан слишком рано, например не были добавлены какие-либо файлы, или перепутан комментарий к коммиту. Если необходимо сделать этот коммит ещё раз, можно выполнить **git 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, приведенная команда была запущена сразу после предыдущего коммита), то состояние проекта будет абсолютно таким же и всё, что измениться, это комментарий к коммиту.

Появится редактор для комментариев к коммитам, но уже с введённым комментарием к последнему коммиту. Здесь можно отредактировать это сообщение так же, как обычно, и оно перепишет предыдущее.

Например, после совершения коммита, оказалось, что было забыто проиндексировать файл, который необходимо включить в данный коммит, необходимо выполнить подобную команду:

```
$ 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

Рассмотрим как переделать изменения в индексе и в рабочем каталоге. Приведём пример. Допустим, внесены изменения в два файла и необходимо записать их как два отдельных коммита, но случайно была выполнена команда **git add *** и проиндексировали оба файла. Как теперь отменить индексацию одного из двух файлов? Команда **git status** выводит подсказку для этого действия:

```
$ 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. Выполним эту команду чтобы отменим 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
#
```

Теперь файл benchmarks.rb изменен, но не в индексе.

Отмена изменений файла

Если возникла ситуация, 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? Команда **git status** также выводит подсказку. 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
#
```

Выполним то что написано в подсказке:

```
$ 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. Необходимо понимать, что это опасная команда: все сделанные изменения в этом файле пропали — просто скопировали поверх него другой файл. Никогда не используйте эту команду, если не полностью уверены, что этот файл не нужен.

Помните, 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 восстановить не получиться.

Просмотр меток

Как и большинство СКВ, git имеет возможность помечать (tag) определённые моменты в истории как важные. Как правило, этот функционал используется для отметки моментов выпуска версий (v1.0, и т.п.).

Просмотр имеющихся меток (tag) в git'е делается просто. Достаточно набрать **git tag**:

```
$ git tag
v0.1
v1.3
```

Данная команда перечисляет метки в алфавитном порядке; порядок их появления не имеет значения.

Для меток также можно осуществлять поиск по шаблону. Например, репозиторий git'a содержит более 240 меток. Если интересует просмотр только выпусков 1.4.2, можно выполнить следующее:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

Создание меток

Git использует два основных типа меток: легковесные и аннотированные. Легковесная метка — это что-то весьма похожее на ветку, которая не меняется — это просто указатель на определённый коммит. А вот аннотированные метки хранятся в базе данных Git'a как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если необходимо сделать временную метку или по какой-то причине нет необходимости сохранять остальную информацию, то для этого годятся и легковесные метки.

Аннотированные метки

Создание аннотированной метки в git'е выполняется легко. Самый простой способ это указать **-a** при выполнении команды **git tag**:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

Опция **-m** задаёт меточное сообщение, которое будет храниться вместе с меткой. Если не указать сообщение для аннотированной метки, git запустит редактор, чтоб ввести его.

Можно посмотреть данные метки вместе с коммитом, который был помечен, с помощью команды **git show**:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97…
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Она показывает информацию о выставившем метку, дату отметки коммита и аннотирующее сообщение перед информацией о коммите.

Легковесные метки

Легковесная метка — это ещё один способ отметки коммитов. В сущности, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесной метки не передаются опций -a, -m:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
```

```
v1.4-lw
v1.5
```

На этот раз при выполнении **git show** на этой метке не будет дополнительной информации. Команда просто покажет помеченный коммит:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97…
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Выставление меток позже

Также возможно помечать уже пройденные коммиты. Предположим, что история коммитов выглядит следующим образом:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Теперь предположим, что было забыто отметить версию проекта v1.2, которая была там, где находится коммит "updated rakefile". Можете добавить метку и позже. Для отметки коммита укажите его контрольную сумму (или её часть) в конце команды:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

Проверим, что коммит теперь отмечен:

```
$ git tag
v0.1
```

```
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
…
```

Перемещение и удаление меток

Для того чтобы переместить метку на другой коммит необходимо использовать опцию **-f**:

```
$ git tag -f v1.2 a6b4c97
```

Для того чтобы удалить метку, необходимо указать опцию **-d** в команде **git tag** с названием метки:

```
$ git tag -d v1.2
```

Порядок выполнения практической работы

1. Изучить теоретическую часть работы.
2. Продолжить работу с созданным репозиторием.
3. Создать три файла: 1.txt, 2.txt, 3.txt.
4. Проиндексировать первый файл и сделать коммит с комментарием “add 1.txt file”.
5. Проиндексировать второй и третий файлы.
6. Удалить из индекса второй файл.

7. Перезаписать уже сделанный коммит с новым комментарием “add 1.txt and 3.txt”

8. Создать аннотированную метку с названием v0.01.

9. Создать легковесную метку v0.01-lw указывающую на первый коммит в репозитории.

10. Удалить метку v0.01.

11. Продемонстрировать преподавателю ход работы, ответить на уточняющие вопросы.

2.5 Практическое занятие «Ветвление. Конфликты»

Цель работы: научиться создавать ветки, перемещаться по ним, объединять и удалять их. Решать конфликты слияния.

Ветвление

Ветка в git'е — это просто легковесный подвижный указатель на один из коммитов. Ветка по умолчанию в git'е называется **master**. Когда происходит создание коммита на начальном этапе, доступна ветка **master**, указывающая на последний сделанный коммит. При каждом новом коммите она сдвигается вперёд автоматически.

Для того чтобы создать новую ветку используется команда **git branch**, создадим ветку с названием **testing**:

```
$ git branch testing
```

Эта команда создаст новый указатель на тот самый коммит, на котором сейчас находится git.

Откуда git узнает, на какой ветке в данный момент находится система контроля версий? Он хранит специальный указатель, который называется HEAD (верхушка). В git'е это указатель на локальную ветку. После выполнения команды **git branch** git всё ещё находиться на ветке **master**. Команда **git branch** только создала новую ветку, она не переключила git на неё.

Чтобы перейти на существующую ветку, необходимо выполнить команду **git checkout**. Перейдём на новую ветку **testing**:

```
$ git checkout testing
```

Изменим какой-нибудь файл и выполним коммит, теперь ветка **testing** передвинулась вперёд, но ветка **master** всё ещё указывает на коммит, на котором git

был, когда выполнялась команда **git checkout**, чтобы переключить ветки. Перейдём обратно на ветку **master**:

```
$ git checkout master
```

Эта команда выполнила два действия. Она передвинула указатель HEAD назад на ветку **master** и вернула файлы в рабочем каталоге назад, в соответствие со снимком состояния, на который указывает **master**. Это также означает, что изменения, которые выполняются, начиная с этого момента, будут ответвляться от старой версии проекта. Это, по сути, откатывает изменения, которые были временно сделаны на ветке testing, так что дальше можно двигаться в другом направлении.

Снова внесем изменения в какой-либо файл и закоммитим их, таким образом произошло разветвление истории проекта. Была создана новая ветка, осуществлен переход на неё, выполнена работа на ней, после чего снова осуществлен переход обратно на основную ветку и выполнена другая работа. Оба эти изменения изолированы в отдельных ветках: можно переключаться туда и обратно между ветками и слить их, когда это потребуется. Всё это осуществляется простыми командами **git branch** и **git checkout**.

Из-за того, что ветка в git'е на самом деле является простым файлом, который содержит 40 символов контрольной суммы SHA-1 коммита, на который он указывает, создание и удаление веток практически беззатратно. Создание новой ветки настолько же быстрое и простое, как запись 41 байта в файл (40 символов + символ новой строки).

Это отличие от того, как в большинстве СКВ делается ветвление. Там это приводит к копированию всех файлов проекта в другой каталог. Это может занять несколько секунд или даже минут, в зависимости от размера проекта, тогда как в git'е это всегда происходит моментально. Также благодаря тому, что git запоминает предков для каждого коммита, поиск нужной базовой версии для слияния уже автоматически выполнен, и в общем случае слияние делается легко. Эти особенности помогают поощрять разработчиков к частому созданию и использованию веток.

Основы слияния

Представим, что разработчик работает над своим проектом и уже имеете пару коммитов. Он решил, что будет работать над проблемой №53 из системы отслеживания ошибок, используемой в его компании. Разумеется, git не привязан к какой-то определенной системе отслеживания ошибок. Так как проблема №53 является обособленной задачей, над которой он собираетесь работать, то он создает

новую ветку, в которой будет работать на ней. Чтобы создать ветку и сразу же перейти на неё, можно выполнить команду **git checkout** с ключом **-b**:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Это сокращение для:

```
$ git branch iss53
$ git checkout iss53
```

Во время работы над своим проектом разработчик делает несколько коммитов. Эти действия сдвигают ветку **iss53** вперёд потому, что он перешел на нее (то есть HEAD указывает на неё).

Теперь он получает звонок о том, что есть проблема с веб-сайтом, которую необходимо немедленно устранить. С git'ом нет нужды делать исправления для неё поверх тех изменений, которые уже сделали в **iss53**, и нет необходимости прикладывать много усилий для отмены этих изменений перед тем, как разработчик сможет начать работать над решением срочной проблемы. Всё, что нужно сделать, это перейти на ветку **master**.

Однако, прежде чем сделать это, необходимо учесть, что если в рабочем каталоге или индексе имеются незафиксированные изменения, которые конфликтуют с веткой, на которую необходимо перейти, git не позволит переключить ветки. Лучше всего при переключении веток иметь чистое рабочее состояние. На данный момент представим, что все изменения были добавлены в коммит, и теперь можно переключиться обратно на ветку **master**:

```
$ git checkout master
Switched to branch "master"
```

Теперь рабочий каталог проекта находится точно в таком же состоянии, что и в момент начала работы над проблемой №53, так что можно сконцентрироваться на исправлении срочной проблемы. Очень важно запомнить: git возвращает рабочий каталог к снимку состояния того коммита, на который указывает ветка, на которую осуществляется переход. Он добавляет, удаляет и изменяет файлы автоматически, чтобы гарантировать, что состояние рабочей копии идентично последнему коммиту на ветке.

Итак, необходимо срочно исправить ошибку. Разработчик создает для этого ветку, на которой будет работать:

```
$ git checkout -b hotfix
```

```
Switched to a new branch "hotfix"
```

Вносит изменения и коммитит их, после чего происходит тестирование. После того как разработчик убедился, что решение работает, необходимо слить (merge) изменения назад в ветку **master**, чтобы включить их в продукт. Это делается с помощью команды **git merge**:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Фраза "Fast forward" в этом слиянии говорит о том, что ветка, которую мы слили, указывала на коммит, являющийся прямым родителем коммита, на котором мы сейчас находимся, git просто сдвинул её указатель вперёд. Иными словами, когда происходит попытка слить один коммит с другим таким, которого можно достигнуть, проследовав по истории первого коммита, git поступает проще, перемещая указатель вперёд, так как нет расходящихся изменений, которые нужно было бы сливать воедино. Это называется "перемотка" (fast forward).

После того как очень важная проблема решена, разработчик готов вернуться обратно к тому, над чем работал перед тем, как его прервали. Однако, сначала необходимо удалить ветку **hotfix**, так как она больше не нужна — ветка **master** уже указывает на то же место. Можете удалить ветку с помощью опции **-d** к **git branch**:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Теперь разработчик может вернуться обратно к рабочей ветке для проблемы №53 и продолжить работать над ней.

Стоит напомнить, что работа, сделанная на ветке **hotfix**, не включена в файлы на ветке **iss53**. Если это необходимо, можно слить ветку **master** в ветку **iss53** посредством команды **git merge master**. Или же можно подождать с интеграцией изменений до тех пор, пока не будет принято решение включить изменения на **iss53** в продуктовую ветку **master**.

Допустим, разработчик разобрался с проблемой №53 и готов объединить эту ветку и свой **master**. Чтобы сделать это, сольём ветку **iss53** в ветку **master** точно так

же, как делали это ранее с веткой **hotfix**. Всё, что необходимо сделать, — перейти на ту ветку, в которую нужно слить изменения, и выполнить команду **git merge**:

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Это слияние немного отличается от слияния, сделанного ранее для ветки **hotfix**. В данном случае история разработки разделилась в некоторой точке. Так как коммит на той ветке, на которой git находится, не является прямым предком для ветки, которую необходимо слить, git'у придётся проделать кое-какую работу. В этом случае git делает простое трёхходовое слияние, используя при этом те два снимка состояния репозитория, на которые указывают вершины веток, и общий для этих двух веток снимок-прародитель.

Теперь, когда слияние наработок осуществлено, ветка **iss53** больше не нужна. Можно удалить её и затем вручную закрыть карточку (ticket) в системе:

```
$ git branch -d iss53
```

Основы конфликтов при слиянии

Иногда процесс слияния не идёт гладко. Если была изменена одна и та же часть файла по-разному в двух ветках, которые необходимо слить, git не сможет сделать это чисто. Если решение проблемы №53 изменяет ту же часть файла, что и **hotfix**, разработчик получит конфликт слияния, и выглядеть он будет примерно так:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git не создал новый коммит для слияния. Он приостановил этот процесс до тех пор, пока конфликт не будет разрешен. Если необходимо посмотреть, какие файлы не прошли слияние (на любом этапе после возникновения конфликта), достаточно выполнить команду **git status**:

```
[master*]$ git status
index.html: needs merge
# 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)
#
#   unmerged:   index.html
#
```

Всё, что имеет отношение к конфликту слияния и что не было разрешено, отмечено как *unmerged*. Git добавляет стандартные маркеры к файлам, которые имеют конфликт, так что можно открыть их вручную и разрешить эти конфликты. Файл с конфликтом содержит секцию, которая выглядит примерно так:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

В верхней части блока (всё что выше =======) это версия из HEAD (ветки **master**, так как именно на неё разработчик перешел перед выполнением команды **git merge**), всё, что находится в нижней части — версия в **iss53**. Чтобы разрешить конфликт, неоходимо либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению. Например, можно разрешить этот конфликт заменой всего блока, показанного выше, следующим блоком:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Это решение содержит понемногу из каждой части, и полностью удалены строки <<<<<<<,======= и >>>>>>>. После того как разработчик разобрался с каждой из таких секций в каждом из конфликтных файлов, необходимо выполнить команду **git add** для каждого конфликтного файла. Индексирование будет означать для git'a, что все конфликты в файле теперь разрешены. Имеется возможность использовать графические инструменты для разрешения конфликтов, необходимо выполнить команду **git mergetool**, которая запустит соответствующий графический инструмент и покажет конфликтные ситуации:

```
$ git mergetool
```

```
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html

Normal merge conflict for 'index.html':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (opendiff):
```

Также имеется возможность выбрать другой инструмент для слияния, отличный от выбираемого по умолчанию (в нашем случае git выбрал opendiff). Все поддерживаемые инструменты, указаны выше, после “merge tool candidates”. Укажите название предпочтительного инструмента с параметром **-t**:

```
$ git mergetool -t meld
```

Но для того, чтобы утилита meld открылась, необходимо задать параметры, чтобы git знал, откуда ее взять. Для этого необходимо выполнить следующую команду:

```
$ git config --global mergetool.meld.path /c/"Program files (x86)"/Meld/meld.exe
```

В качестве последнего параметра команды, необходимо указать путь до программы в системе.

После того как будет осуществлен выход из инструмента для выполнения слияния, git спросит, было ли оно успешным. Если ответ будет да — файл индексируется (добавляется в область для коммита), что показывает, что конфликт разрешён.

Выполним команду **git status** ещё раз, чтобы убедиться, что все конфликты были разрешены:

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

Если конфликт разрешен, разработчик удостоверился, что всё, имевшее конфликты, было проиндексировано, необходимо выполнить **git commit** для

завершения слияния. По умолчанию сообщение коммита будет выглядеть примерно так:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

Также можно дополнить это сообщение информацией о том, как был разрешен конфликт, так как это может быть полезно для других в будущем.

Порядок выполнения практической работы

1. Изучить теоретическую часть работы.
2. Продолжить работу с созданным репозиторием.
3. Создать новую ветку my_first_branch.
4. Перейти на ветку и создать новый файл in_branch.txt, закоммитить изменения.
5. Вернуться на ветку master.
6. Создать и сразу перейти на ветку new_branch.
7. Сделать изменения в файле 1.txt, добавить строчку “new row in 1.txt file”, закоммитить изменения.
8. Перейти на ветку master и слить изменения с ветки my_first_branch, после чего слить изменения с ветки new_branch.
9. Удалить ветки my_first_branch и new_branch.
10. Создать ветку branch_1 и branch_2.
11. Перейти на ветку branch_1 и изменить файл 1.txt, удалить все содержимое и добавить текст “fix in 1.txt”, изменить файл 3.txt, удалить все содержимое и добавить текст “fix in 3.txt”, закоммитить изменения.
12. Перейти на ветку branch_2 и также изменить файл 1.txt, удалить все содержимое и добавить текст “My fix in 1.txt”, изменить файл 3.txt, удалить все содержимое и добавить текст “My fix in 3.txt”, закоммитить изменения.
13. Слить изменения ветки branch_2 в ветку branch_1.

14. Решить конфликт файла 1.txt в ручном режиме, а конфликт 3.txt используя команду git mergetool с утилитой Meld.

15. Продемонстрировать преподавателю ход работы, ответить на уточняющие вопросы.

3 МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОРГАНИЗАЦИИ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

3.1 Общие положения

Целями самостоятельной работы являются систематизация, расширение и закрепление теоретических знаний в области различных аспектов управления жизненным циклом программных систем.

Самостоятельная работа студента по дисциплине «Управление жизненным циклом программных систем» включает следующие виды деятельности:

1) проработка лекционного материала;

2) подготовка к практическим занятиям;

4) выполнение контрольной работы;

3) подготовка к экзамену.

В ходе самостоятельной работы студент, ориентируясь на изложенные рекомендации, планирует свое время и перечень необходимых работ в зависимости от индивидуальных психофизических особенностей. Формат самостоятельной работы студентов может отличаться в зависимости от формы обучения и объема аудиторной работы.

3.2 Проработка лекционного материала и подготовка к практическим занятиям

Для качественного усвоения учебного материала целесообразно осуществлять проработку лекционного материала, которая направлена как на систематизацию имеющегося материала, так и на подготовку к освоению практических аспектов, связанных с содержанием дисциплины.

Проработка лекционного материала включает деятельность, связанную с изучением рекомендуемых преподавателем источников, в которых отражены основные моменты, затрагиваемые в ходе лекций. Кроме того, важное место отведено работе с собственноручно составленным конспектом лекций. При конспектировании во время лекции помните, что не следует записывать все, что говорит и/или демонстрирует лектор: старайтесь выявить главное и записать только это. Цель конспекта – формирование целостного логически выстроенного взгляда на круг вопросов, затрагиваемых в ходе изучения соответствующей темы, а не механическая фиксация текстовой и графической информации.

Во внеаудиторное время проработка лекционного материала может быть выстроена в двух основных форматах:

а) отработка прослушанной лекции (прочтение конспекта и рекомендованных преподавателем источников с сопоставлением записей) и восполнение пробелов, если они имелись (например, если студент не понял чего-то, не успел записать);

б) прочтение перед каждой последующей лекцией предыдущей, дабы не тратилось много времени на восстановление контекста изучения дисциплины при продолжающейся или связанной теме.

В ходе проработки лекционного материала обращайте внимание на контрольные вопросы, которые, как правило, имеются в конце каждой темы учебника (учебного пособия). Отвечая на них, можно сделать вывод о степени понимания материала. Если ответы на какие-то вопросы вызвали затруднения, то следует предпринять еще одну попытку изучения отдельных вопросов.

При подготовке к практическому занятию необходимо заранее изучить методические рекомендации по его проведению, обратить внимание на цель, формат и содержание занятия. Если какие-то моменты вызвали дополнительные вопросы, целесообразно обратиться к содержанию лекционного материала, рекомендациям преподавателя по изучению теоретической части курса (рекомендуемым источникам) или за личной консультацией. В ходе подготовки к лабораторным работам может потребоваться обращение к различным источникам. Проявляйте инициативу и самостоятельность в данном вопросе. При этом следует пользоваться только авторитетными изданиями, как печатными, так и электронными.

3.3 Контрольная работа «Обзор современной системы / методологии, помогающей управлять жизненным циклом»

Студенту предлагается написать реферат о современной системе или методологии, помогающей управлять жизненным циклом программных систем.

Написание реферата является формой самостоятельной деятельности студента, позволяющей раскрыть его способности по работе с литературными и (или) иными источниками.

Реферат должен содержать обзор источников, посвященных выбранной теме. Реферирование не должно сводиться к простому переписыванию исходных текстов. В ходе работы следует осуществлять критическое осмысление основных положений, сопоставление и интерпретацию позиций различных авторов.

При написании реферата необходимо в равной степени пользоваться различными источниками: учебниками (учебными пособиями), статьями из научных изданий, авторитетными электронными ресурсами, а также другими источниками.

На все источники должны быть ссылки по тексту работы.

Реферат должен состоять из введения, основной части и заключения. Во введении обосновывается актуальность темы, формулируются цель, задачи, объект и предмет исследования. Основная часть должна иметь иерархическую структуру: состоять из разделов и подразделов. В заключении сжато, но емко по содержанию указываются основные результаты, полученные при написании реферата.

Темы для реферата:

1) Обзор системы управления проектами Jira.
2) Обзор Service Desk системы Freshservice.
3) Обзор гибкой методологии управления Agile.
4) Обзор системы контроля версий SVN.
5) Обзор системы контроля версий Git.
6) Обзор системы контроля версий Mercurial.
7) Обзор системы управления проектами Redmine.
8) Обзор методологии Git Flow.
9) Обзор сравнения системы контроля версий Git и SVN.
10) Обзор визуальных клиентов для системы контроля версий Git.
11) Обзор Service Desk системы Kayako.
12) Обзор Service Desk системы TechDesk.
13) Обзор Service Desk системы Itsm 365.

3.4 Подготовка к экзамену

Рекомендации по подготовке к экзамену

Для проведения экзамена составляются билеты. В состав билета входят 2 теоретических вопроса и одна практическая задача (определяющая умение применить практические знания на конкретном примере).

В рамках проработки экзаменационных вопросов, настоятельно рекомендуется пользоваться различными источниками информации, а не ограничиваться «базовым» ресурсом. Важно использовать актуальные источники информации (современные учебники и т.п.), что обусловлено постоянным обновлением и совершенствованием технологий.

Перечень теоретических вопросов для проведения экзамена

1) Распределённые рабочие процессы.
2) Каскадная модель.
3) Каскадная модель с промежуточным контролем.
4) Спиральная модель.

5) Инкрементная модель.

6) Модель разработки через тестирование (V-модель).

7) Эволюционная модель.

8) Этапы создания ПС: формирование требований, концептуальное проектирование, спецификация приложений, разработка моделей, интеграция и тестирование программных систем.

9) Организация планирования жизненного цикла ПС.

10) Структура планов жизненного цикла ПС.

11) Задачи планов для обеспечения жизненного цикла ПС.

12) Git Flow, описание методологии, достоинства и недостатки.

13) Перемещение, описание механизма, основное отличие от слияния.

14) История СКВ, основные виды, описание и характеристика, достоинства и недостатки.

15) История Git. Основное отличие от других СКВ. Описание состояний файлов в Git.

16) Agile, описание методологии, характеристика, достоинства и недостатки.

17) SCRUM, описание фреймворка, характеристика, достоинства и недостатки.

18) Метки, предназначение, описание и характеристика.

19) Ветвление в Git, механизм работы, сравнение с другими СКВ.

20) Слияние веток, описание стратегий слияния, конфликты.

Вариант практической задачи для проведения экзамена

Перед вами команды работы программиста с системой контроля версии git. Необходимо расшифровать что было проделано. Пояснить каждую строку скрипта.

```
$ git remote -v
$ git pull my_server
$ git checkout -b somefix
$ cat ‘fix’ >> 1_bug_file.php
$ git commit -a -m “fix big problem”
$ git checkout master
$ git merge somefix
$ git log -1 -p
$ git push my_server master
$ git branch sometest
$ git checkout sometest
$ cat ‘fix 2’ >> 1_bug_file.php
$ git add 1_bug_file.php
```

```
$ git diff --cached
$ git commit
$ git checkout master
$ git merge sometest
$ git branch -d sometest
```

4 Рекомендуемые источники

Зараменских, Е. П. Управление жизненным циклом информационных систем : учебник и практикум для академического бакалавриата [Электронный ресурс] / Е. П. Зараменских. — М. : Издательство Юрайт, 2018. — 431 с. Режим доступа: https://biblio-online.ru/viewer/258E13A0-41F6-4A48-AE82-2EF782B29F96/upravlenie-zhiznennym-ciklom-informacionnyh-sistem#/

Ехлаков, Ю. П. Организация бизнеса на рынке программных продуктов: Учебник [Электронный ресурс] / Ю. П. Ехлаков. — Томск: ТУСУР, 2012. — 314 с. — Режим доступа: https://edu.tusur.ru/publications/970

Ехлаков, Ю. П. Управление программными проектами: Учебник [Электронный ресурс] / Ю. П. Ехлаков. — Томск: ТУСУР, 2015. — 217 с. — Режим доступа: https://edu.tusur.ru/publications/6024

Ехлаков, Ю. П. Экономика программной инженерии : Учебное пособие [Электронный ресурс] / Ю. П. Ехлаков. — Томск: ТУСУР, 2013. — 132 с. — Режим доступа: https://edu.tusur.ru/publications/4527

Ехлаков, Ю. П. Теоретические основы автоматизированного управления: Учебник [Электронный ресурс] / Ю. П. Ехлаков. — Томск: ТУСУР, 2001. — 338 с. — Режим доступа: https://edu.tusur.ru/publications/668