

Лабораторная работа №3

Основы Git и GitHub. Регулярные выражения

Цель работы: ознакомиться с основами системы контроля версий Git, научиться работать с регулярными выражениями.

Необходимое ПО для практической части: JDK 8; IntelliJ IDEA 14 Community Edition; Git.

Теоретические сведения

О контроле версий

Что такое контроль версий, и зачем он вам нужен? **Система контроля версий (СКВ)** — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. Программисты обычно помещают в систему контроля версий исходные коды программ, но на самом деле под версионный контроль можно поместить файлы практически любого типа.

Если вы графический или веб-дизайнер и хотели бы хранить каждую версию изображения или макета, то пользоваться системой контроля версий будет очень мудрым решением. СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Вообще, если, пользуясь СКВ, вы всё испортите или потеряете файлы, всё можно будет легко восстановить. Вдобавок, накладные расходы будут очень маленькими.

Локальные системы контроля версий

Многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы.

Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов (см. рис. 1.1).

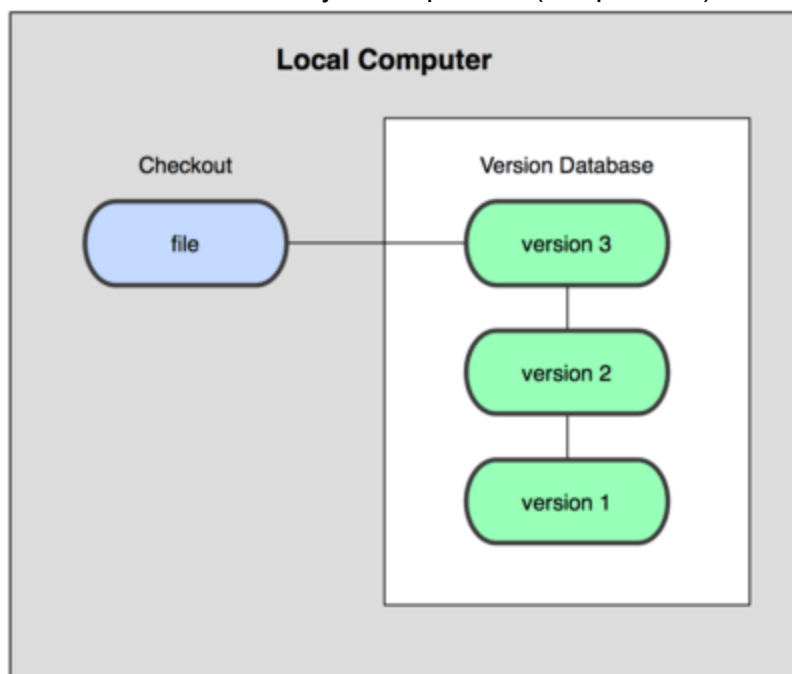


Рисунок 1.1. Схема локальной СКВ

Одной из наиболее популярных СКВ такого типа является *rcs*, которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита *rcs* устанавливается вместе с Developer Tools. Эта утилита основана на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами), которые хранятся в специальном

формате на диске. Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи.

Централизованные системы контроля версий

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например *CVS*, *Subversion* и *Perforce*, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий (см. рис. 1.2).

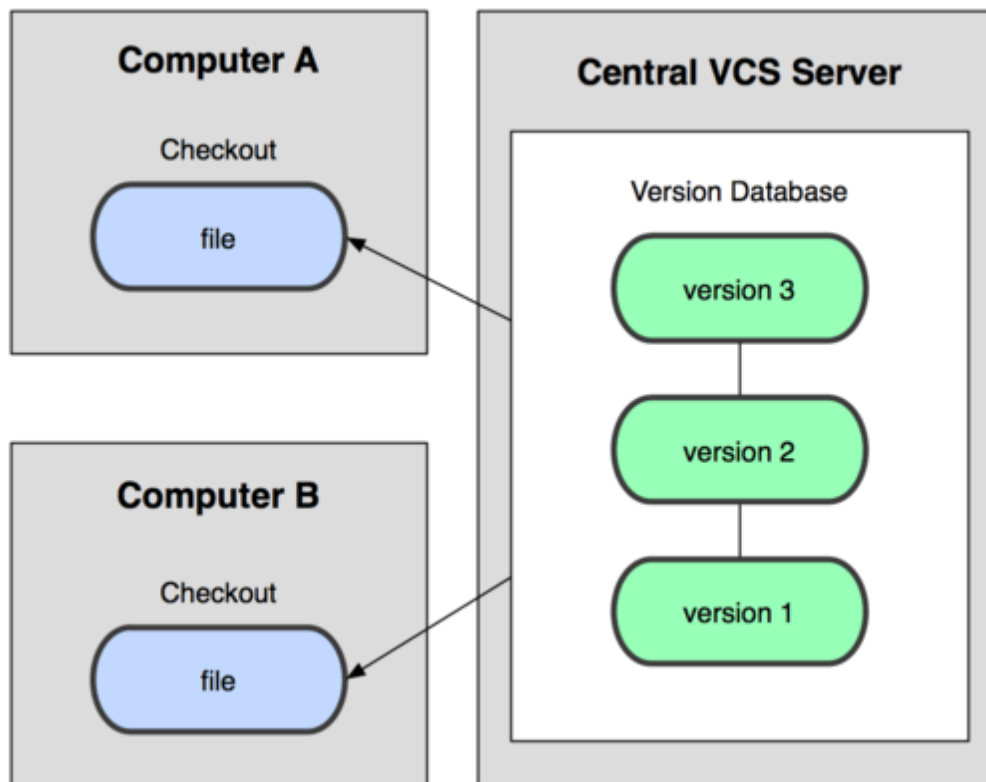


Рисунок 1.2. Схема централизованного контроля версий

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как *Git*, *Mercurial*, *Bazaar* или *Darcs* клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий (репозиторий - место, где хранятся и поддерживаются какие-либо данные, в данном случае, данные, находящиеся под версионным контролем). Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных (см. рис. 1.3).

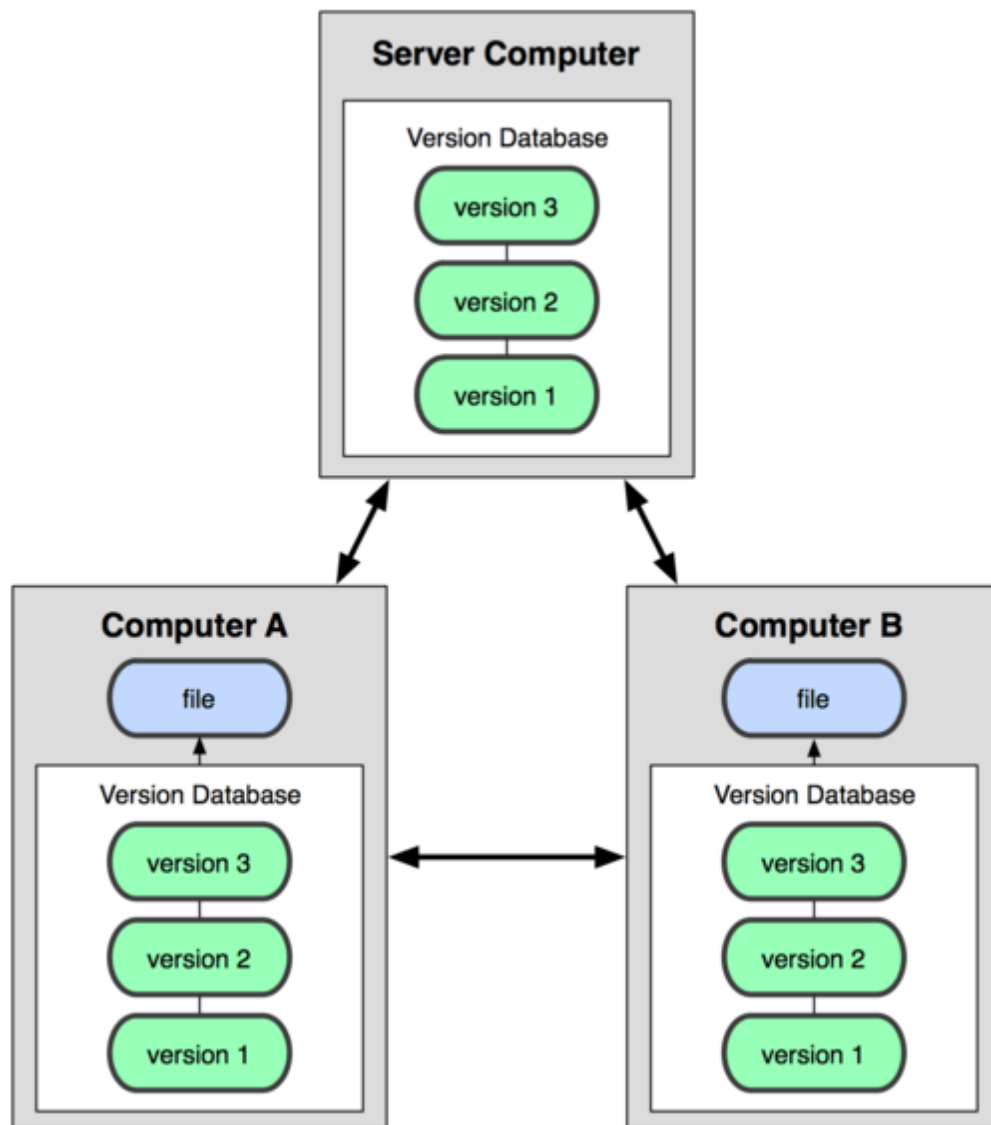


Рисунок 1.3. Схема распределённой системы контроля версий

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

Основы Git

Так что же такое Git в двух словах? Эту часть важно усвоить, поскольку если вы поймёте, что такое Git, и каковы принципы его работы, вам будет гораздо проще пользоваться им эффективно. Изучая Git, постарайтесь освободиться от всего, что вы знали о других СКВ, таких как Subversion или Perforce. В Git'e совсем не такие понятия об информации и работе с ней как в других системах, хотя пользовательский интерфейс очень похож. Знание этих различий защитит вас от путаницы при использовании Git'a.

Слепки вместо патчей

Главное отличие Git'a от любых других СКВ (например, Subversion и ей подобных) — это то, как Git смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени, как показано на рисунке 1.4.

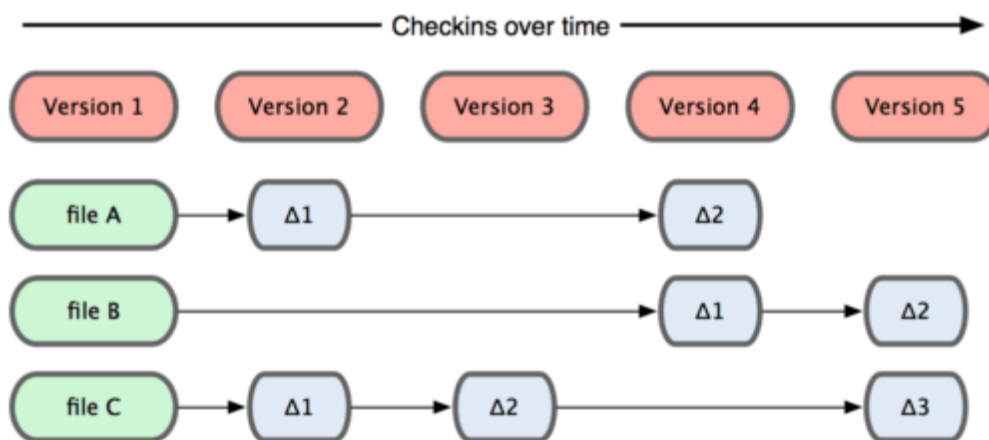


Рисунок 1.4. Другие системы хранят данные как изменения к базовой версии для каждого файла

Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных, похоже на рисунок 1.5.

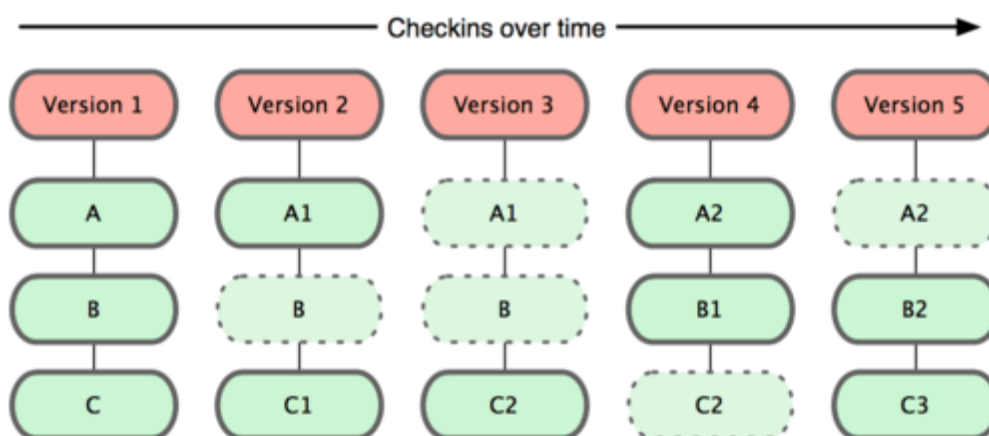


Рисунок 1.5. Git хранит данные как слепки состояний проекта во времени

Это важное отличие Git'a от практически всех других систем контроля версий. Из-за него Git вынужден пересмотреть практически все аспекты контроля версий, которые другие системы переняли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто СКВ.

Почти все операции — локальные

Для совершения большинства операций в Git'e необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.

К примеру, чтобы показать историю проекта, Git'у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у СКВ-сервера или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или VPN. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а VPN-клиент не работает, всё равно можно продолжать работать. Во многих других системах это невозможно или же крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория).

Git следит за целостностью данных

Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git'a и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм, используемый Git'ом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в Git'e на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

24b9da6552252987aa493b52f8696cd6d3b00373

Работая с Git'ом, вы будете встречать эти хеши повсюду, поскольку он их очень широко использует. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого.

Чаще всего данные в Git только добавляются

Практически все действия, которые вы совершаете в Git'e, только добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой СКВ, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их очень сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.

Три состояния

Это самое важное, что нужно помнить про Git, если вы хотите, чтобы изучение шло гладко. В Git'e файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих Git, есть три части: каталог Git'a (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).

Local Operations

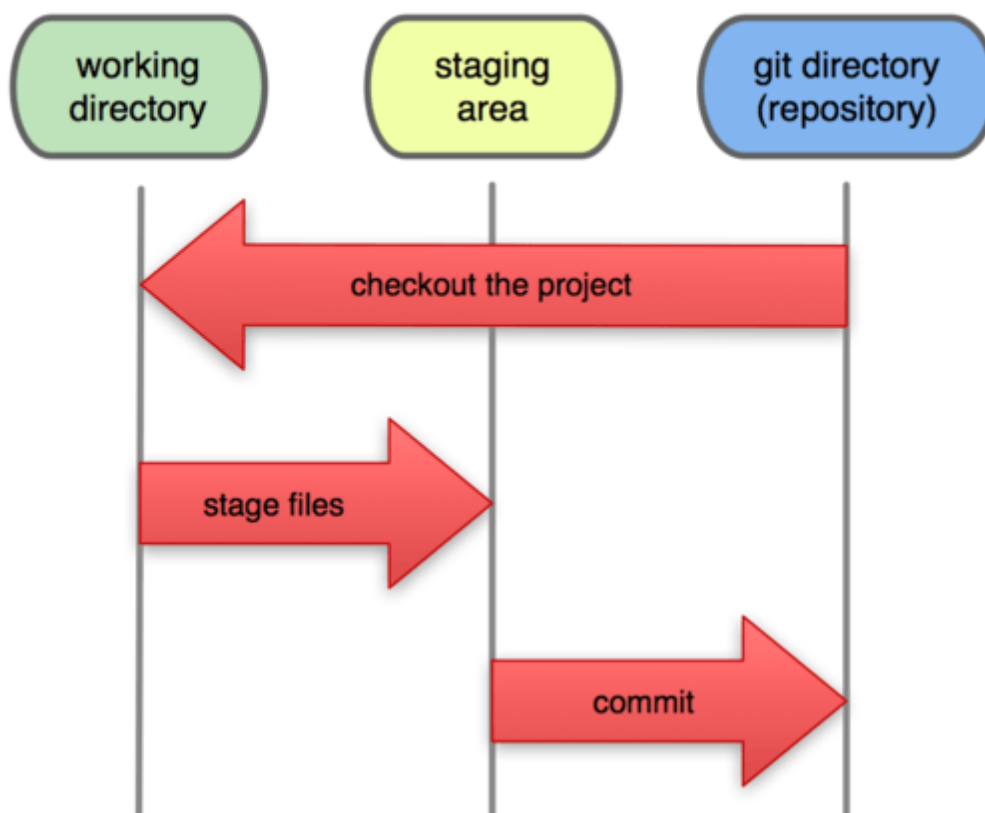


Рисунок 1.6. Рабочий каталог, область подготовленных файлов, каталог Git'a

Каталог Git'a — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git'a, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git'a и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git'a, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Стандартный рабочий процесс с использованием Git'a выглядит примерно так:

1. Вы вносите изменения в файлы в своём рабочем каталоге.
2. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git'a на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git'a, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

GitHub

GitHub — самый крупный веб-сервис для хостинга IT-проектов и их совместной разработки. Основан на системе контроля версий Git и разработан компанией GitHub, Inc (ранее Logical Awesome).

Сервис абсолютно бесплатен для проектов с открытым исходным кодом и предоставляет им все возможности, а для частных проектов предлагаются различные платные тарифные планы.

Регулярные выражения

Регулярное выражение — это маска или образец последовательности символов. Они используются для поиска и изменения подстрок в тексте. В качестве примеров типичных задач, которые решаются с использованием регулярных выражений, можно привести:

- валидация строки, которая должна представлять собой ip адрес, email или логин;
- замена устаревшего названия на новое;
- подсчет того, сколько раз некоторая конструкция встречается в тексте;
- нахождение в директории файлов с определенным расширением, например, txt.

Таблица регулярных выражений (шпаргалка)

выражение	значение
.	любой символ
аб	последовательность символов
a б	любой из символов
[абв]	любое из значений в списке
[^абв]	все, кроме значений в списке
[a-г]	любое из значений в промежутке
[a-гA-Г]	любой из промежутков
^выражение	начало строки
выражение\$	конец строки

\d	цифра
\D	все, кроме цифр
\s	пробельный символ, эквивалентно [\t\n\r]
\S	все, кроме пробельных символов
выражение{количество}	сколько раз встречается, например, a{2}
выражение{мин,макс}	сколько раз встречается, промежуток
выражение{мин,}	сколько раз встречается, минимум
выражение*	0 или более раз
выражение+	1 или более раз
выражение?	0 или 1 раз
(выражение)	использовать группу как единое выражение, например, (ab)+

Основные элементы регулярных выражений

Метасимволы

Большинство символов не изменяют свое значение в регулярном выражении. Например, выражения «ю» или «123» выражают сами себя, то есть строки текста «ю» и «123». Однако, если бы регулярные выражения обозначали только сами себя, от них не было бы большой пользы. Для обозначения сразу группы символов используются метасимволы, например:

- «.» — обозначает любой символ
- \d — обозначает любой символ цифры
- \s — обозначает пробельные символы, например, пробел или табуляция

Из-за того что «.» является метасимволом, для обозначения обычного символа в тексте её и другие метасимволы нужно экранировать с помощью косой черты, «\». С символами «d» и «s» ситуация обратная, без косой черты они значат сами себя, а с ней становятся метасимволами. Может возникнуть вопрос, а как же быть с самой косой чертой «\»? Для использования косой черты как саму себя её нужно выделять косой чертой. Еще больше осложняет ситуацию то, что в Java, «\» в строке, тоже является специальным символом, для того чтобы передать его методам работы с регулярными выражениями, его тоже нужно выделить косой чертой. Например, регулярное выражение обозначающее цифру «\d» в Java строке будет выглядеть «String regExp = «\\d». Обратите внимание на это “удвоение” числа косых черт при составлении регулярного выражения.

Начало и конец строки

Еще два полезных метасимвола обозначают позицию подстроки в строке:

- ^ — обозначает начало строки
- \$ — обозначает конец строки

Например, регулярное выражение «^d» обозначает цифру в самом начале строки, а «\.\$» обозначает строки оканчивающиеся точкой.

Последовательность и выбор

- a\d — последовательность, сначала символ «a», а потом цифра
- a|d — выбор, или символ «a», или цифра

Перечисления и промежутки и отрицание

- [абв] — обозначает любой из символов внутри квадратных скобок, «а», «б» или «в»
- [а-г] — обозначает любое из значений в промежутке от «а» до «г» то есть «а», «б», «в» или «г»
- [а-ГА-Г] — обозначает любой из символов в промежутках от «а» до «г» и от «А» до «Г»

- `[^абв]` — любые символы кроме «а», «б» и «в»

Группы

Группа позволяет оперировать сразу с целой конструкцией, она обозначается с помощью круглых скобок `()`, например, `«(2\d)»` обозначает группу из двух цифр, первая из которых является двойкой. Группы (а также отдельные символы) можно использовать как выражения в квантификаторах, о которых пойдет речь далее.

Квантификаторы

Квантификаторы позволяют указывать сколько раз повторяется выражение

- `выражение{количество}` — указать сколько раз повторяется выражение, например, `(01){3}` будет сокращением для `«010101»`
- `выражение{мин,макс}` — указать промежуток сколько раз повторяется выражение, например `(010){1,3}` обозначает `«010»`, `«0100101»` или `«010010010»`
- `выражение{мин,}` — указать только минимальное количество, максимальное не ограничено
- `выражение*` — 0 или более раз
- `выражение+` — 1 или более раз
- `выражение?` — 0 или один раз

Жадные, ленивые и собственнические квантификаторы

Эти определения выражают то, как будет осуществляться поиск регулярного выражения, что необходимо для того, чтобы разрешить возникающие неоднозначности.

- Жадные квантификаторы начинают с того что сопоставляют с выражением как можно большую подстроку. Если это не приводит к успеху, то они на каждом следующем шаге уменьшают свой «аппетит». Жадность является состоянием по умолчанию для квантификаторов, поэтому нам не нужно её специально указывать.
- Ленивые квантификаторы начинают с как можно меньшей подстроки, постепенно увеличивая её, пока не добьются успеха. Для их обозначения используется символ `«?»` после квантификатора, например, `«выражение*?»` или `«выражение{3}?»`.
- Собственнические квантификаторы, как и жадные начинают с наибольшей подстроки, но, в отличие от жадных, на этом и заканчивают. Если сопоставление с наибольшей подстрокой не привело к разрешению регулярного выражения то возвращается несовпадение. Собственнические квантификаторы обозначаются с помощью символа `«+»`, например, `«выражение++»` или `«выражение{1,2}+»`.

Рассмотрим пример. Пусть у нас есть строка `«00183295200»`. Рассмотрим регулярное выражение `«. *00»`, оно использует жадный квантификатор и найдет в нашей подстроке являющуюся всей строкой, т.е. `«00183295200»`. Ленивое регулярное выражение `«. *?00»` вернет `«00»`. Собственническое же выражение `«. *+00»` не вернет вообще ничего, `«. *»` заберет все символы и `«00»` сопоставить не удастся.

Обратные ссылки

Круглые скобки помимо логического отделения выражений играют ещё одну роль, а именно создают т.н. группы. Они полезны, когда ваше регулярное выражение состоит из нескольких одинаковых частей. Тогда достаточно описать один раз однотипную часть шаблона, а потом сослаться на неё.

Пример:

```
public static void main(String[] args){
    Pattern p = Pattern.compile("(якороль).+(\\1)");
    Matcher m = p.matcher("регулярные выражения это круто регулярные
    выражения это круто регулярные выражения это круто якороль Я СЕГОДНЯ ЕЛ БАНАНЫ
    якороль регулярные выражения это круто");
    if(m.find()){
        System.out.println(m.group());
    }
}
```

Результатом будет:

якороль Я СЕГОДНЯ ЕЛ БАНАНЫ якороль

На месте первой группы (якороль) могло содержаться более сложное выражение, тогда обратная ссылка \1 значительно сократила бы размер регулярного выражения.

Группы нумеруются слева направо, начиная с 1. Каждая открывающая скобка увеличивает номер группы:

```
(  (  )  )(  (  )  )
^  ^      ^  ^
1  2      3  4
```

Нулевая группа совпадает со всей найденной подпоследовательностью.

Классы для работы с регулярными выражениями

Java классы, предназначенные для работы с регулярными выражениями это *java.util.regex.Pattern* и *java.util.regex.Matcher*. Однако некоторые операции с регулярными выражениями можно выполнить с помощью методов класса *java.util.String*:

- `boolean matches(String regexp)` — проверяет, удовлетворяет ли строка регулярному выражению
- `String replaceFirst(String regexp, String replacement)` — заменяет первую найденную подстроку удовлетворяющую выражению `regexp` на `replacement`
- `String replaceAll(String regexp, String replacement)` — заменяет все подстроки удовлетворяющие выражению `regexp` на `replacement`
- `String[] split(String regex)` — разбивает строку в массив, разделителями служат подстроки, удовлетворяющие регулярному выражению
- `String[] split(String regex, int limit)` — то же, что и метод выше, плюс позволяет установить предел разбиения

Класс *java.util.regex.Pattern* предназначен для хранения скомпилированного регулярного выражения. Объекты этого класса создаются с помощью статического метода `compile`:

```
Pattern p = Pattern.compile("\\d+");
```

Также он содержит метод `matcher` для создания объекта класса *java.util.regex.Matcher*:

```
Matcher m = p.matcher("a120a0bs");
```

Основные методы класса *Matcher*:

- `boolean find()` — найти следующую подстроку, удовлетворяющую выражению
- `boolean find(int start)` — вариация предыдущего метода, начинает поиск заново, а также позволяет указать с какого по счету символа начать
- `String group()` — возвращает предыдущую найденную подстроку (группу)
- `String group(int group)` — возвращает найденную подстроку по индексу
- `int groupCount()` — возвращает количество найденных подстрок
- `int start()` — возвращает начальную позицию предыдущей найденной подстроки
- `int start(int group)` — вариация предыдущего метода, позволяющая указать индекс подстроки (группы)
- `int end()` — возвращает конечную позицию предыдущей найденной подстроки
- `int end(int group)` — вариация предыдущего метода, позволяющая указать индекс подстроки (группы)
- `Matcher reset()` — очищает поиск, позволяет начать все заново
- `Matcher reset(CharSequence input)` — вариация предыдущего метода позволяющая поменять строку в которой ведется поиск
- `Matcher usePattern(Pattern newPattern)` — указать новый паттерн

Пример, найдем в строке цифр подстроки, состоящие из единиц и выведем начальные и конечные позиции этих подстрок в исходной строке:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```

public class FindOnes {

    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("1+");
        String s = "302130032111239021130";
        Matcher matcher = pattern.matcher(s);
        while(matcher.find()){
            System.out.println("found: " + matcher.group());
            System.out.println("start position: " + matcher.start());
            System.out.println("end position: " + matcher.end());
            System.out.println();
        }
    }
}

```

Программа выведет следующее:

```

found: 1
start position: 3
end position: 4

```

```

found: 111
start position: 9
end position: 12

```

```

found: 11
start position: 17
end position: 19

```

Практическая часть

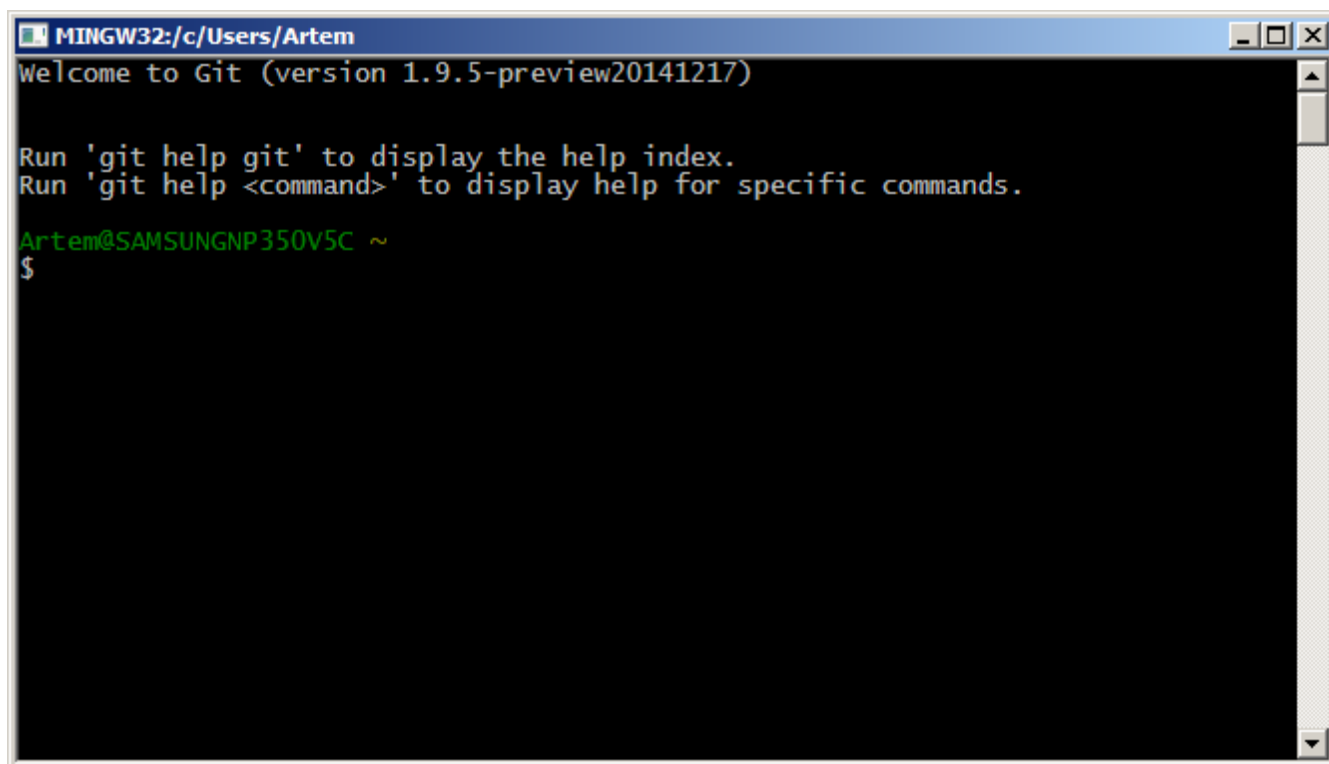
С системой контроля версий можно работать многими способами: далее будет рассмотрена работа с Git через консоль и через IntelliJ IDEA.

Установка в Windows

Установить Git в Windows очень просто. У проекта msysGit процедура установки — одна из самых простых. Просто скачайте exe-файл инсталлятора со страницы проекта на GitHub'e и запустите его:

<http://msysgit.github.com/>

После установки у вас будет как консольная версия, так и стандартная графическая.



```

MINGW32:/c/Users/Artem
Welcome to Git (version 1.9.5-preview20141217)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Artem@SAMSUNGNP350V5C ~
$

```

Пожалуйста, при необходимости использовать консольную версию Git, используйте командную оболочку Git Bash, входящую в состав msysGit, потому что так вы сможете запускать сложные команды, приведённые в примерах в настоящей книге. Командная оболочка Windows использует иной синтаксис, из-за чего примеры в ней могут работать некорректно.

Первоначальная настройка Git

Теперь, когда Git установлен в вашей системе, необходимо настроить среду для работы с Git'ом под себя. Это нужно сделать только один раз — при обновлении версии Git'a настройки сохранятся. Но вы можете поменять их в любой момент, выполнив те же команды снова.

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

Имя пользователя

Первое, что вам следует сделать после установки Git'a, — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git'e содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Проверка настроек

Если вы хотите проверить используемые настройки, можете использовать команду `git config --list`, чтобы показать все, которые Git найдёт:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
...
```

Как получить помощь?

Если вам нужна помощь при использовании Git'a, есть два способа открыть страницу руководства по любой команде Git'a:

```
$ git help <команда>
$ git <команда> --help
```

Например, так можно открыть руководство по команде `config`:

```
$ git help config
```

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

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.

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд `git add` указывающих индекслируемые файлы, а затем `commit`:

```
$ git add *.c
```

```
$ git add README
```

```
$ git commit -m 'initial project version'
```

Мы разберём, что делают эти команды чуть позже. На данном этапе, у вас есть Git-репозиторий с добавленными файлами и начальным коммитом.

Клонирование существующего репозитория

Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда `git clone`. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется `clone`, а не `checkout`. Это важное отличие — 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`. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования.

Клонирование репозитория осуществляется командой `git clone [url]`. Например, если вы хотите клонировать библиотеку Ruby Git, известную как Grit, вы можете сделать это следующим образом:

```
$ 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 вы можете использовать. В предыдущем примере использовался протокол `git://`, вы также можете встретить `http(s)://` или `user@server:/path.git`, использующий протокол передачи SSH.

Запись изменений в репозиторий

Итак, у вас имеется Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать “снимки” состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

File Status Lifecycle

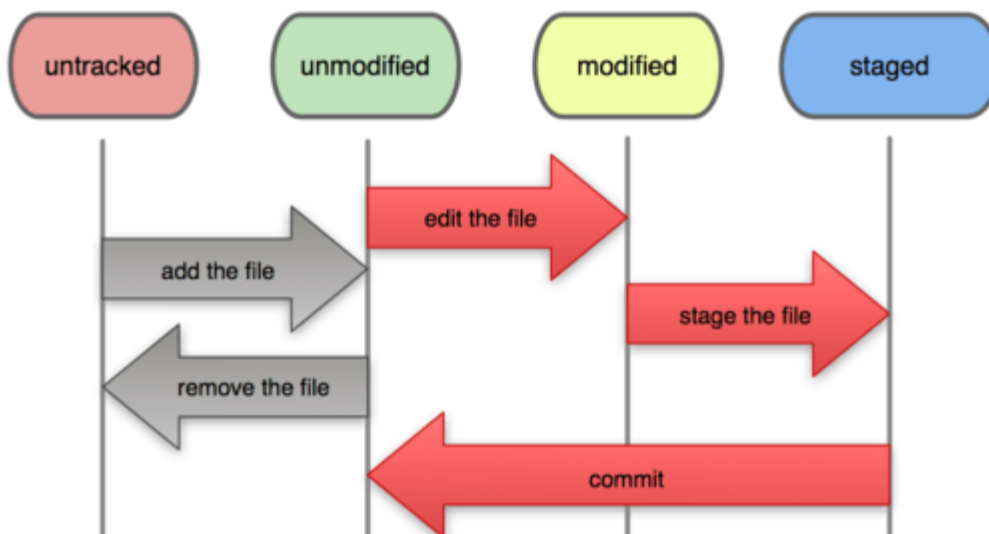


Рисунок 2.1. Жизненный цикл состояний файлов

Запомните, каждый файл в вашем 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 вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что вы только взяли их из хранилища (checked them out) и ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, т.к. вы изменили их 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.

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

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

Это означает, что у вас 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 вам на какой ветке (branch) вы сейчас находитесь. Пока что это всегда ветка `master` — это ветка по умолчанию; в этой лабораторной работе это не важно.

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 вы выполните команду `status`, то увидите, 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
#
```

Вы можете видеть, что файл проиндексирован по тому, что он находится в секции “Changes to be committed”. Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды `git add`, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили `git init`, вы затем выполнили `git add` (файлы) — это было сделано для того, чтобы добавить файлы в ваш каталог под версионный контроль. Команда `git add` принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет (индексирует) все файлы в данном каталоге.

Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл `benchmarks.rb` и после этого снова выполните команду `status`, то результат будет примерно 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). Выполним `git add`, чтобы проиндексировать `benchmarks.rb`, а затем снова выполним `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 вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в `benchmarks.rb` до фиксации. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте ещё раз выполним `git status`:

```
$ 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. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду git add. Если вы выполните коммит сейчас, то файл benchmarks.rb попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду git add, а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения git commit. Если вы изменили файл после выполнения git add, вам придётся 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
#
```

Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите 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, вы можете создать файл .gitignore с перечислением шаблонов соответствующих таким файлам. Вот пример файла .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. Вы можете также включить каталоги log, tmp или pid; автоматически создаваемую документацию; и т.д. и т.п. Хорошая практика заключается в настройке файла .gitignore до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

Фиксация изменений

Теперь, когда ваш индекс настроен так, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили git add после момента редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли git status, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать git commit:

```
$ git commit
```

Эта команда откроет выбранный вами текстовый редактор. (Редактор устанавливается системной переменной \$EDITOR — обычно это vim или emacs, хотя вы можете установить ваш любимый с помощью команды git config --global core.editor).

В редакторе будет отображён следующий текст (это пример окна Vim'a):

```
# 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
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы ("выхлоп") команды `git status` и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете. (Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент `-v` в команду `git commit`. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом вы сможете точно увидеть всё, что сделано.) Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением (удаляя комментарии и вывод diff'a).

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой `commit`, указав его после параметра `-m`, как в следующем примере:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 README
```

Итак, вы создали коммит! Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (`master`), какая контрольная сумма SHA-1 у этого коммита (`463dc4f`), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и остается в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание 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 вам обойтись без `git add`:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
1 files changed, 5 insertions(+), 0 deletions(-)
```

Обратите внимание на то, что в данном случае перед коммитом вам не нужно выполнять `git add` для файла `benchmarks.rb`.

Просмотр истории коммитов

После того как вы создадите несколько коммитов, или же вы клонируете репозиторий с уже существующей историей коммитов, вы, вероятно, захотите оглянуться назад и узнать, что же 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, выполните:

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

В результате выполнения `git log` в данном проекте, вы должны получить что-то вроде этого:

```
$ 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 вы можете видеть, эта команда отображает каждый коммит вместе с его контрольной суммой SHA-1, именем и электронной почтой автора, датой создания и комментарием.

Существует превеликое множество параметров команды `git log` и их комбинаций, для того чтобы показать вам именно то, что вы ищете.

Отмена изменений

На любой стадии может возникнуть необходимость что-либо отменить. Здесь мы рассмотрим несколько основных инструментов для отмены произведённых изменений. Будьте осторожны, ибо не всегда можно отменить сами отмены. Это одно из немногих мест в Git'e, где вы можете потерять свою работу если сделаете что-то неправильно.

Изменение последнего коммита

Одна из типичных отмен происходит тогда, когда вы делаете коммит слишком рано, забыв добавить какие-то файлы, или напутали с комментарием к коммиту. Если вам хотелось бы сделать этот коммит ещё раз, вы можете 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, вы запустили приведённую команду сразу после предыдущего коммита), то состояние проекта будет абсолютно таким же и всё, что вы измените, это комментарий к коммиту.

Появится всё тот же редактор для комментариев к коммитам, но уже с введённым комментарием к последнему коммиту. Вы можете отредактировать это сообщение так же, как обычно, и оно переписшет предыдущее.

Для примера, если после совершения коммита вы осознали, что забыли проиндексировать изменения в файле, которые хотели добавить в этот коммит, вы можете сделать что-то подобное:

```
$ git commit -m 'initial commit'
```

```
$ git add forgotten_file
```

```
$ git commit --amend
```

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

Работа с удалёнными репозиториями

Чтобы иметь возможность совместной работы над каким-либо Git-проектом, необходимо знать, как управлять удалёнными репозиториями. Удалённые репозитории — это модификации проекта, которые хранятся в интернете или ещё где-то в сети. Их может быть несколько, каждый из которых, как правило, доступен для вас либо только на чтение, либо на чтение и запись. Совместная работа включает в себя управление удалёнными репозиториями и помещение (push) и получение (pull) данных в и из них тогда, когда нужно обменяться результатами работы. Управление удалёнными репозиториями включает умение добавлять удалённые репозитории, удалять те из них, которые больше не действуют, умение управлять различными удалёнными ветками и определять их как отслеживаемые (tracked) или нет и прочее. Данный раздел охватывает все перечисленные навыки по управлению удалёнными репозиториями.

Отображение удалённых репозитория

Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду `git remote`. Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы клонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, `origin` — это имя по умолчанию, которое Git присваивает серверу, с которого вы клонировали:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

Чтобы посмотреть, какому URL соответствует сокращённое имя в Git, можно указать команде опцию `-v`:

```
$ git remote -v
origin git://github.com/schacon/ticgit.git (fetch)
origin git://github.com/schacon/ticgit.git (push)
```

Если у вас больше одного удалённого репозитория, команда покажет их все. Например, мой репозиторий `Grit` выглядит следующим образом.

```
$ cd grit
$ git remote -v
bakkdoor git://github.com/bakkdoor/grit.git
cho45 git://github.com/cho45/grit.git
defunkt git://github.com/defunkt/grit.git
koke git://github.com/koke/grit.git
origin git@github.com:mojombo/grit.git
```

Это означает, что мы легко можем получить изменения от любого из этих пользователей.

Добавление удалённых репозитория

В предыдущих разделах мы упомянули и немного продемонстрировали добавление удалённых репозитория, сейчас мы рассмотрим это более детально. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, выполните `git remote add [сокращение] [url]`:

```
$ git remote
origin
```

```
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb      git://github.com/paulboone/ticgit.git
```

Теперь вы можете использовать в командной строке имя `pb` вместо полного URL. Например, если вы хотите извлечь (`fetch`) всю информацию, которая есть в репозитории Павла, но нет в вашем, вы можете выполнить `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]      master    -> pb/master
* [new branch]      ticgit     -> pb/ticgit
```

Ветка `master` Павла теперь доступна локально как `pb/master`. Вы можете слить (`merge`) её в одну из своих веток или перейти на эту ветку, если хотите её проверить.

Fetch и Pull

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

```
$ git fetch [имя удал. сервера]
```

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты.

Когда вы клонируете репозиторий, команда `clone` автоматически добавляет этот удалённый репозиторий под именем `origin`. Таким образом, `git fetch origin` извлекает все наработки, отправленные (`push`) на этот сервер после того, как вы клонировали его (или получили изменения с помощью `fetch`). Важно отметить, что команда `fetch` забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду `git pull`. Она автоматически извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку. Этот способ может для вас оказаться более простым или более удобным. К тому же по умолчанию команда `git clone` автоматически настраивает вашу локальную ветку `master` на отслеживание удалённой ветки `master` на сервере, с которого вы клонировали (подразумевается, что на удалённом сервере есть ветка `master`). Выполнение `git pull`, как правило, извлекает (`fetch`) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (`merge`) их с кодом, над которым вы в данный момент работаете.

Push

Когда вы хотите поделиться своими наработками, вам необходимо отправить (`push`) их в главный репозиторий. Команда для этого действия простая: `git push [удал. сервер] [ветка]`. Чтобы отправить вашу ветку `master` на сервер `origin` (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду `push`. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду `push`, а затем команду `push` выполняете вы, то ваш `push` точно будет отклонён. Вам придётся сначала вытянуть (`pull`) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить `push`.

Инспекция удалённого репозитория

Если хотите получить побольше информации об одном из удалённых репозиториях, вы можете использовать команду `git remote show [удал. сервер]`. Если вы выполните эту команду с некоторым именем, например, `origin`, вы получите что-то подобное:

```
$ git remote show origin
* remote origin
URL: git://github.com/schacon/ticgit.git
Remote branch merged with 'git pull' while on branch master
  master
Tracked remote branches
  master
  ticgit
```

Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке `master`, выполните `git pull`, ветка `master` с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации, и наверняка вы встретились с чем-то подобным. Однако, если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от `git remote show`:

```
$ git remote show origin
* remote origin
URL: git@github.com:defunkt/github.git
Remote branch merged with 'git pull' while on branch issues
  issues
Remote branch merged with 'git pull' while on branch master
  master
New remote branches (next fetch will store in remotes/origin)
  caching
Stale tracking branches (use 'git remote prune')
  libwalker
  walker2
Tracked remote branches
  acl
  apiv2
  dashboard2
  issues
  master
  postgres
Local branch pushed with 'git push'
  master:master
```

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении `git push`. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере. И для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении `git pull`.

Удаление и переименование удалённых репозиториях

Для переименования ссылок в новых версиях Git'a можно выполнить `git remote rename`, это изменит сокращённое имя, используемое для удалённого репозитория. Например, если вы хотите переименовать `pb` в `paul`, вы можете сделать это следующим образом:

```
$ git remote rename pb paul
$ git remote
origin
```

paul

Стоит упомянуть, что это также меняет для вас имена удалённых веток. То, к чему вы обращались как pb/master, стало paul/master.

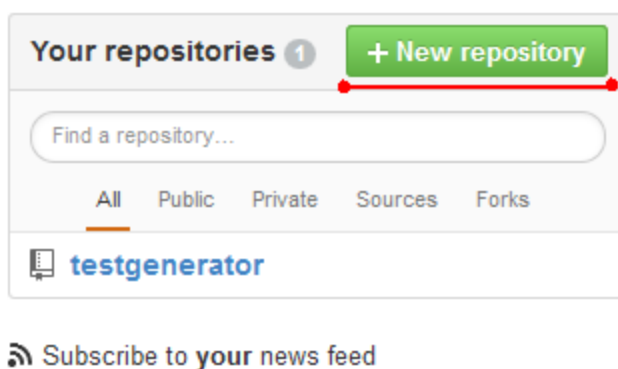
Если по какой-то причине вы хотите удалить ссылку (вы сменили сервер или больше не используете определённое зеркало, или, возможно, контрибьютор перестал быть активным), вы можете использовать `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

Работа над проектом с использованием Git, Github и IntelliJ IDEA

Создание репозитория на GitHub

Для возможности работать с удалёнными репозиториями на GitHub необходима регистрация. После регистрации необходимо создать новый репозиторий.



Отметьте необходимые параметры для начала работы: адрес репозитория, приватность и начальную инициализацию.

Owner: earring / Repository name: ✓

Great repository names are short and memorable. Need inspiration? How about **laughing-dangerzone**.

Description (optional)

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▼

Add a license: **None** ▼ ⓘ

Create repository

В итоге вам откроется страница с содержимым репозитория.

earring / regexp_lab

Unwatch 1

Star 0

Fork 0

Description

Short description of this repository

Website

Website for this repository (optional)

Save or Cancel

1 commit

1 branch

0 releases

1 contributor

branch: master regexp_lab / +

Initial commit

earring authored just now

latest commit 3a91fb8a91

README.md

Initial commit

just now

README.md

regexp_lab

содержимое репозитория

адрес репозитория

<> Code

Issues 0

Pull requests 0

Wiki

Pulse

Graphs

Settings

HTTPS clone URL

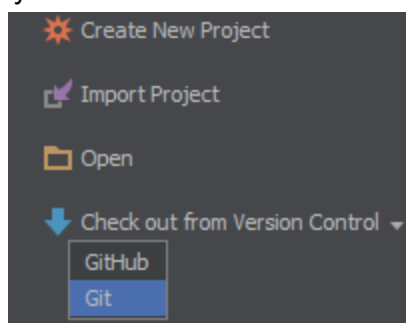
https://github.com/earring/regexp_lab

You can clone with HTTPS, SSH, or Subversion.

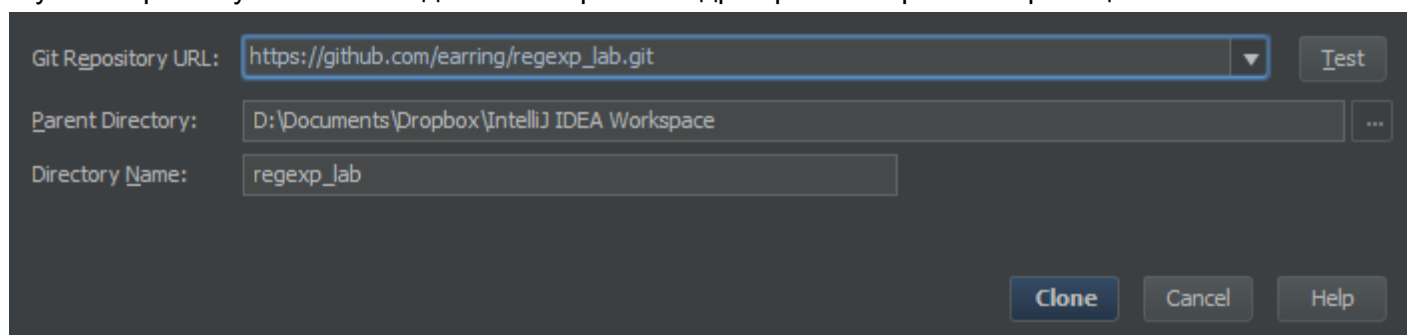
Clone in Desktop

Download ZIP

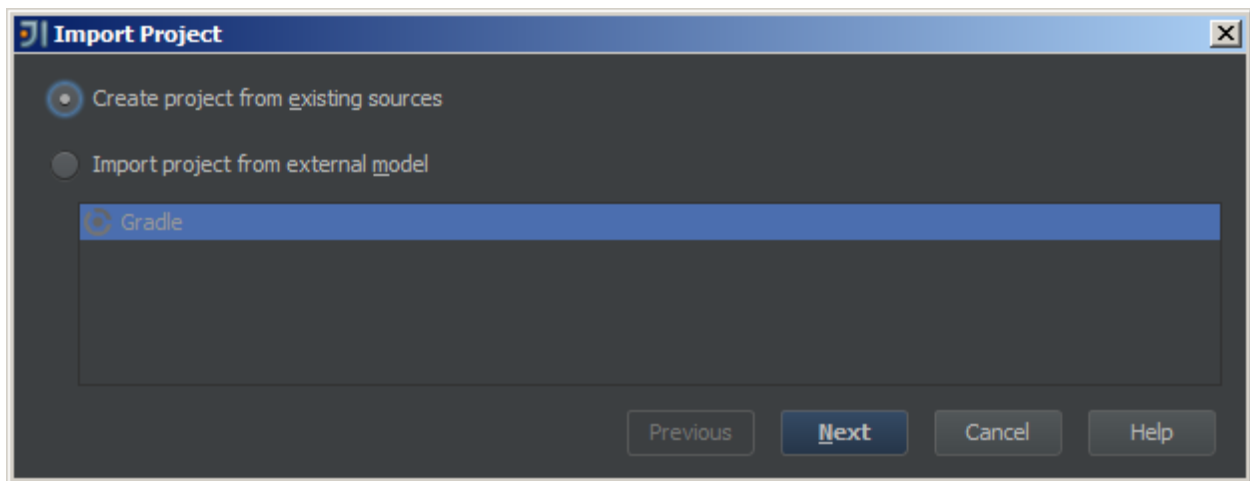
После запуска IntelliJ IDEA там необходимо закрыть открытый проект (последний запускавшийся проект), и в начальном меню выбрать пункт "Check out from Version Control" -> Git.



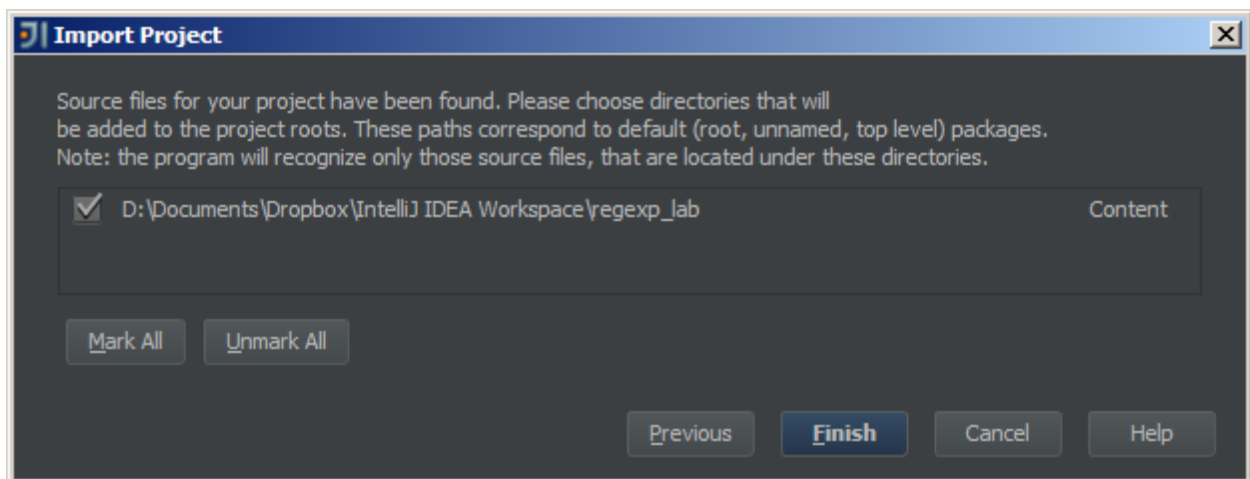
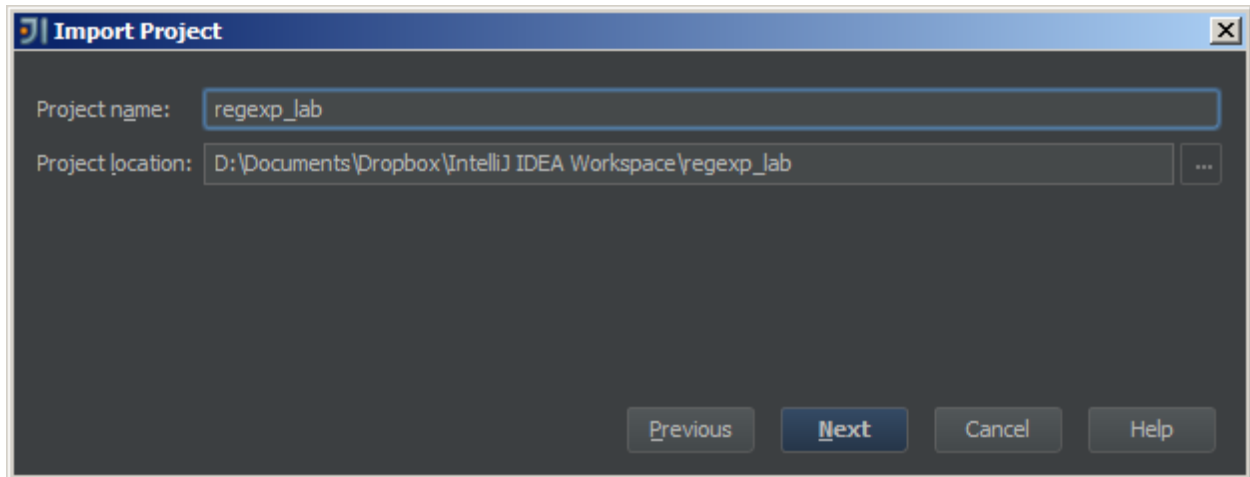
После этого необходимо настроить параметры доступа к только что созданному репозиторию. В строку Git Repository URL необходимо скопировать адрес репозитория со страницы GitHub.



"Clone". Далее надо согласиться с созданием проекта в IntelliJ IDEA, а также задать источник создания проекта, в данном случае - это "existing sources".



Далее идут настройка имени проекта и выбор директорий для проекта.



После этого проект (пока пустой) будет открыт в среде разработки.

Во встроенном терминале (в составе нижних вкладок) можно выполнять непосредственно консольные программы. Если команда `git` не выполняется, то проверьте наличие в системной переменной `Path` пути к папке, где лежат исполняемые файлы `Git`. При установке по умолчанию это `C:\Program Files (x86)\Git\bin`.

```

Terminal
+
D:\Documents\Dropbox\IntelliJ IDEA Workspace\regex_lab>git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

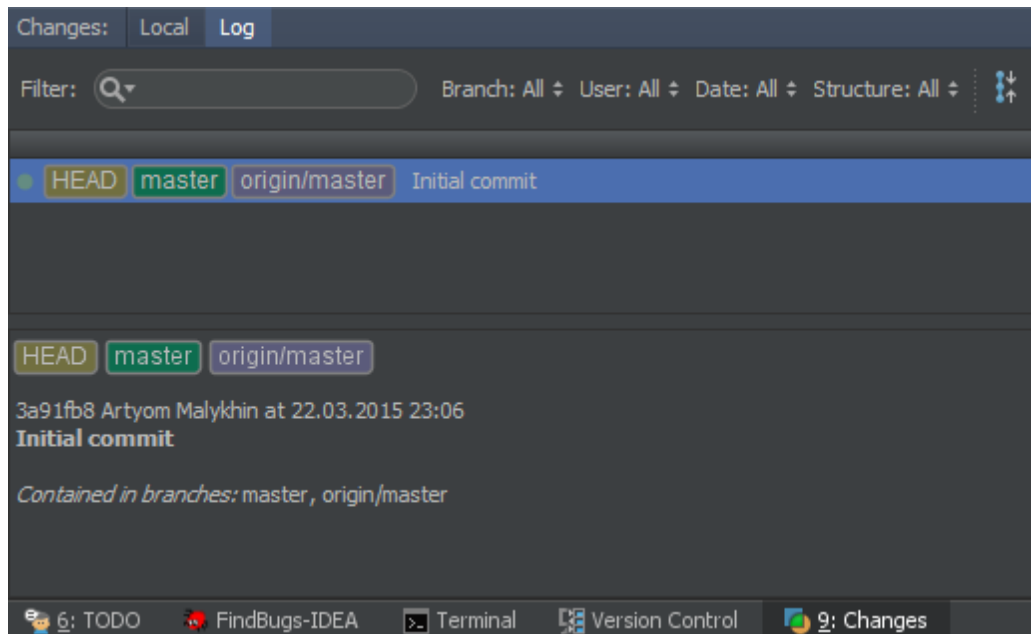
        .idea/
        regex_lab.iml

nothing added to commit but untracked files present (use "git add" to track)

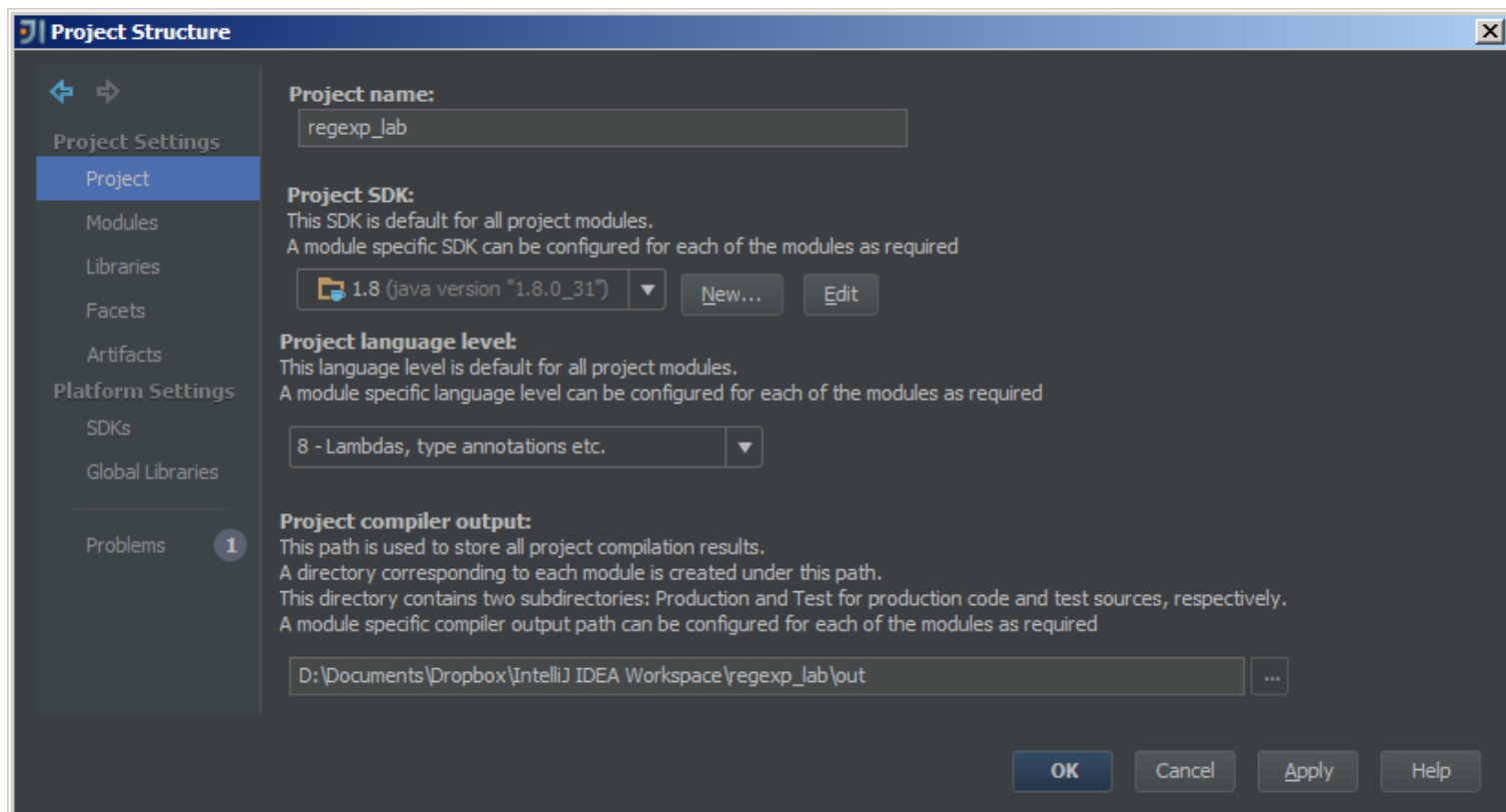
D:\Documents\Dropbox\IntelliJ IDEA Workspace\regex_lab>

```

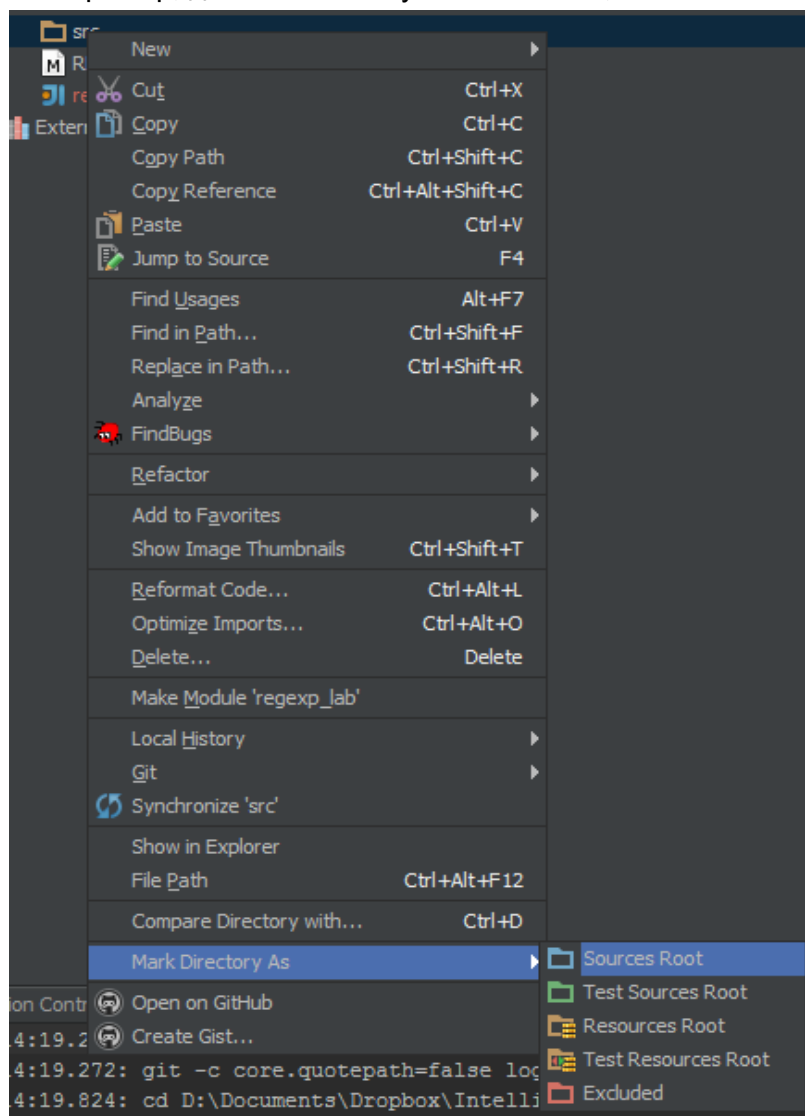
Во вкладке “Changes” можно просмотреть историю изменений в локальном репозитории, например, полученные коммиты. На рисунке изображен начальный коммит (пока он только один).



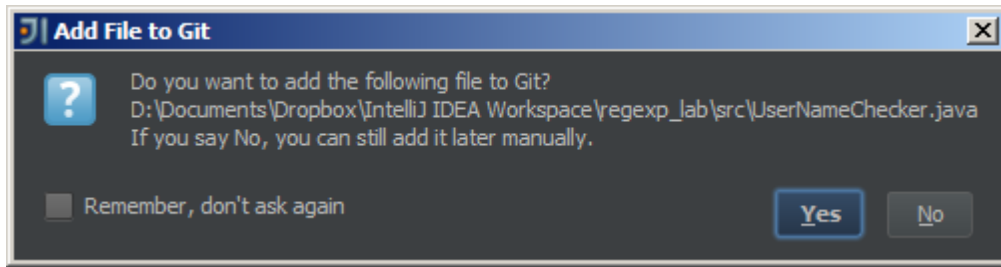
Для начала работы над проектом нужно настроить работу проекта с Java. В контекстном меню проекта - “Open Module Settings”. Настройки должны быть как на скриншоте.



Для хранения исходных кодов создайте папку src. Далее необходимо разметить папки, какую роль каждая из них выполняет. Например, для папки src нужно отметить, что в ней находятся исходные файлы.



После этого станет возможно создать исходные файлы Java из контекстного меню. После создания Java файла, среда разработки предложит добавить этот файл в Git.



В качестве примера приведен исходный файл простой программы, демонстрирующий проверку логина на соответствие условиям “длина от 3 до 15 символов, может включать в себя латинские символы в нижнем регистре, цифры, символ подчеркивания и тире”.

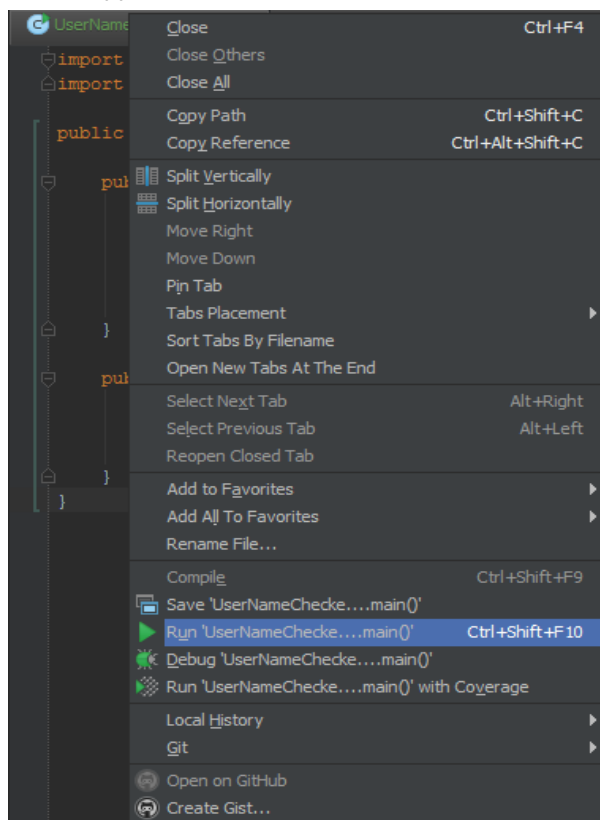
```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class UserNameChecker {

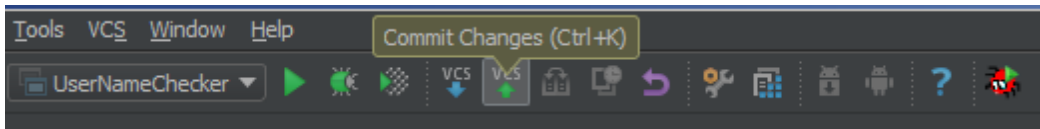
    public static void main(String[] args){
        System.out.println("Results of checking:");
        System.out.println(checkWithRegExp("_@BEST"));
        System.out.println(checkWithRegExp("vovan"));
        System.out.println(checkWithRegExp("vo"));
        System.out.println(checkWithRegExp("Z@OZA"));
    }

    public static boolean checkWithRegExp(String userNameString){
        Pattern p = Pattern.compile("^[a-z0-9_-]{3,15}$");
        Matcher m = p.matcher(userNameString);
        return m.matches();
    }
}
```

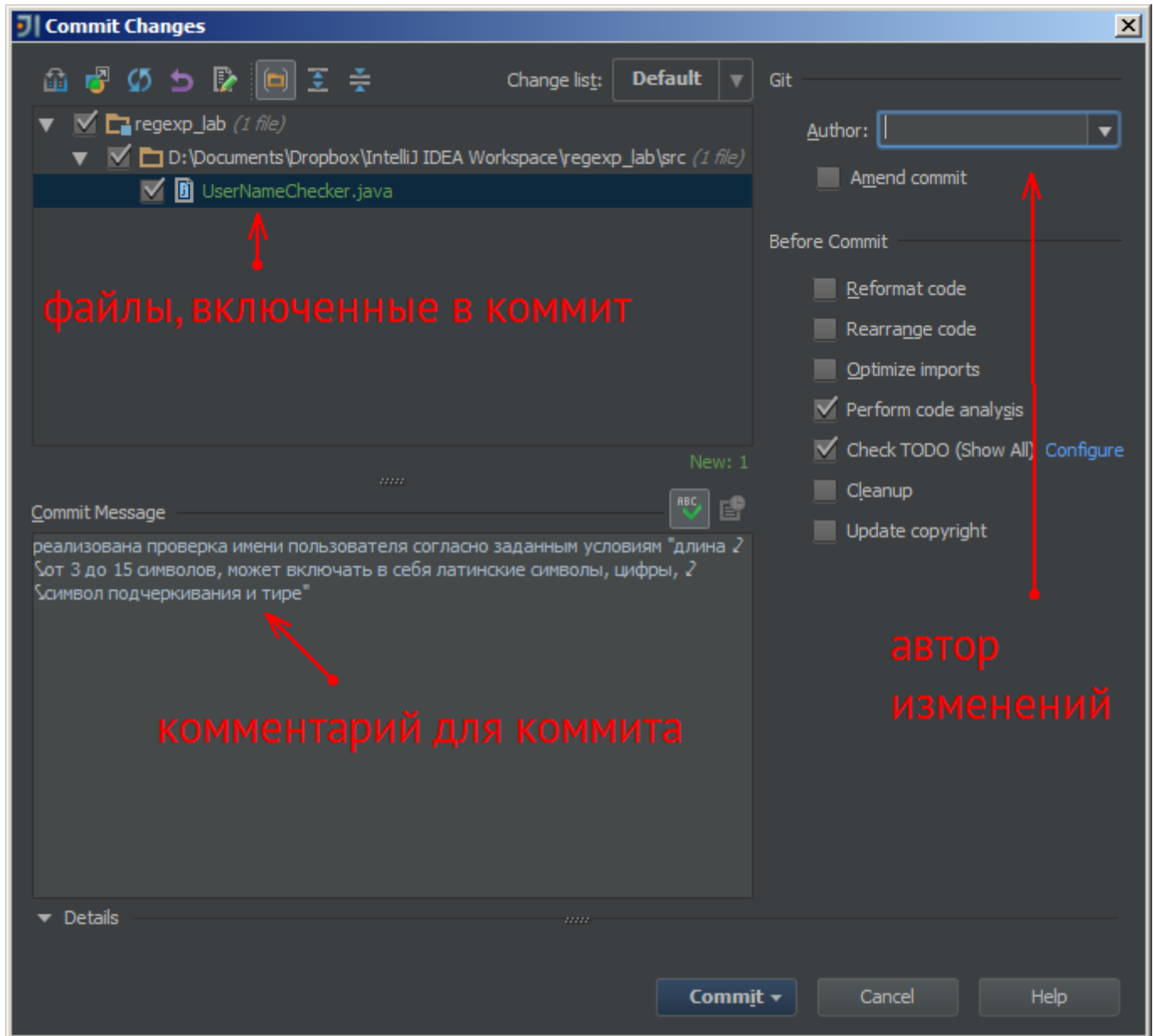
Запустить на исполнение программу можно посредством контекстного меню на вкладке Java файла, в котором находится main-метод.



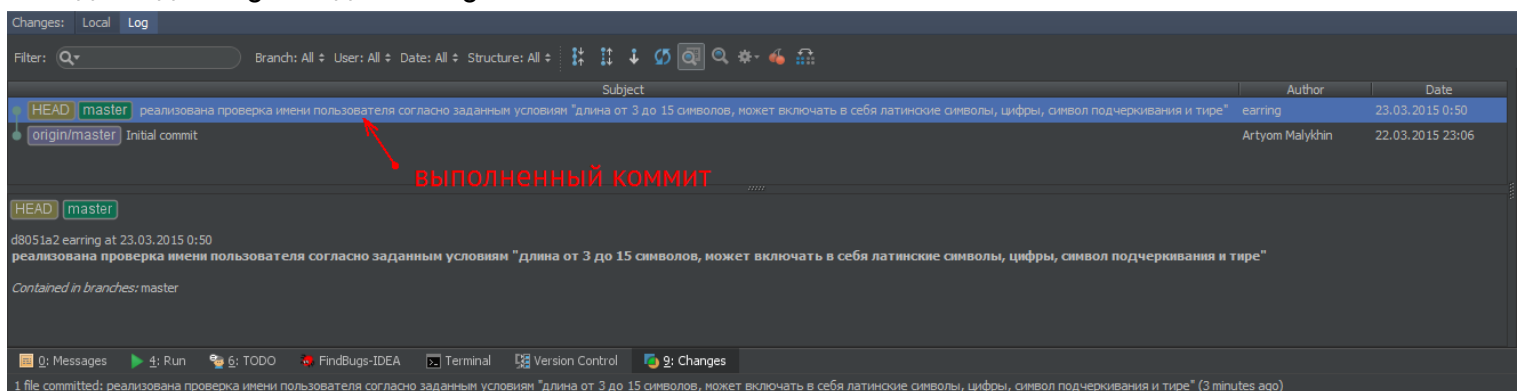
После успешной проверки первой реализации нужно отправить коммит, т.е зафиксировать свою реализацию. Выберите пункт меню “Commit changes”.



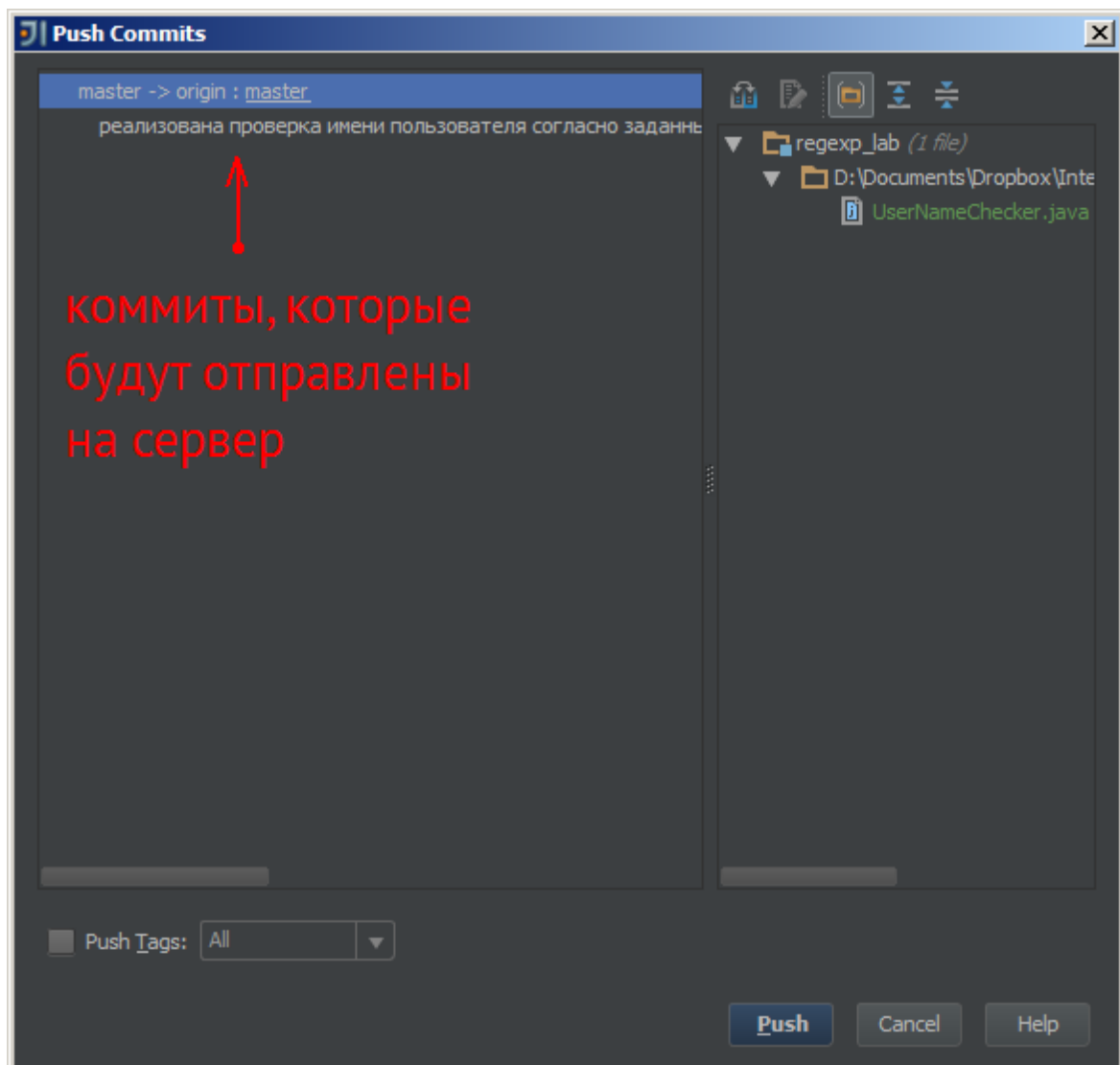
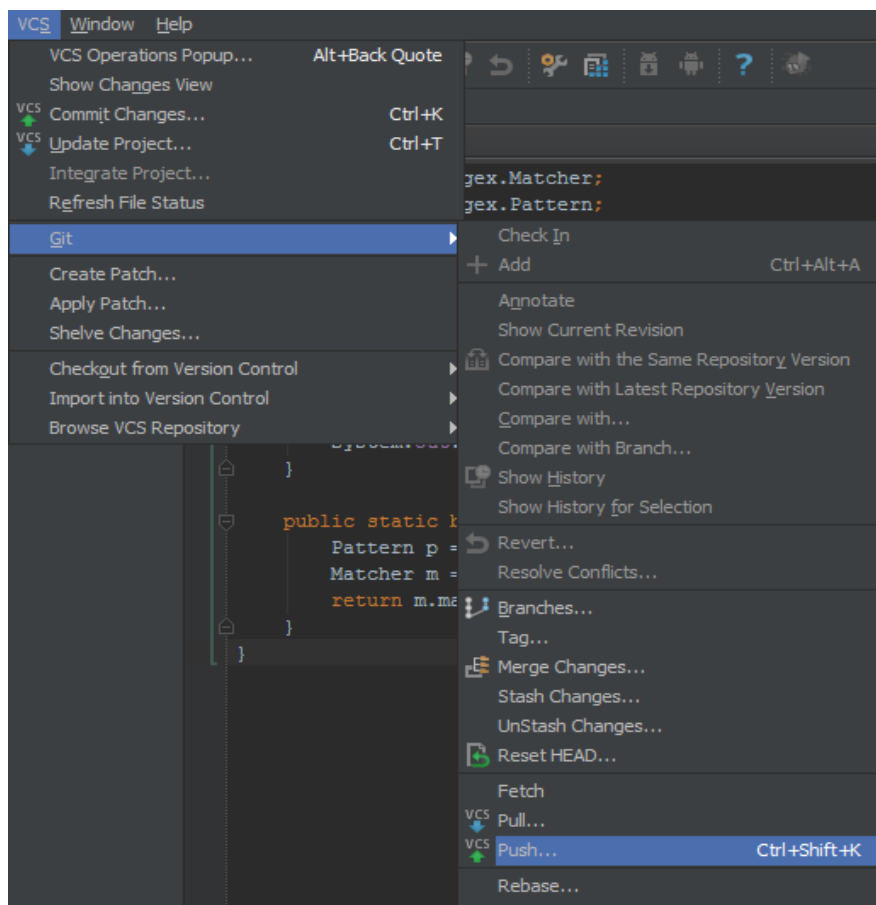
После этого будет показано окно, где можно “оформить” коммит.



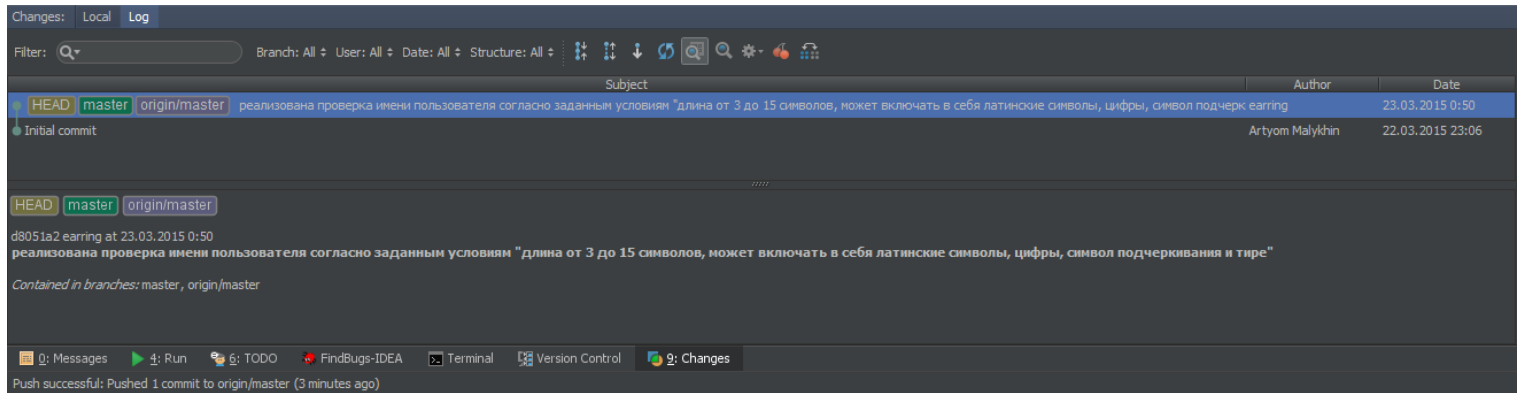
После нажатия “Commit” изменения будут отмечены в локальном репозитории, что можно увидеть во подвкладке Log вкладки Changes.



Теперь эти изменения можно отправить на GitHub. Это делается следующим образом.

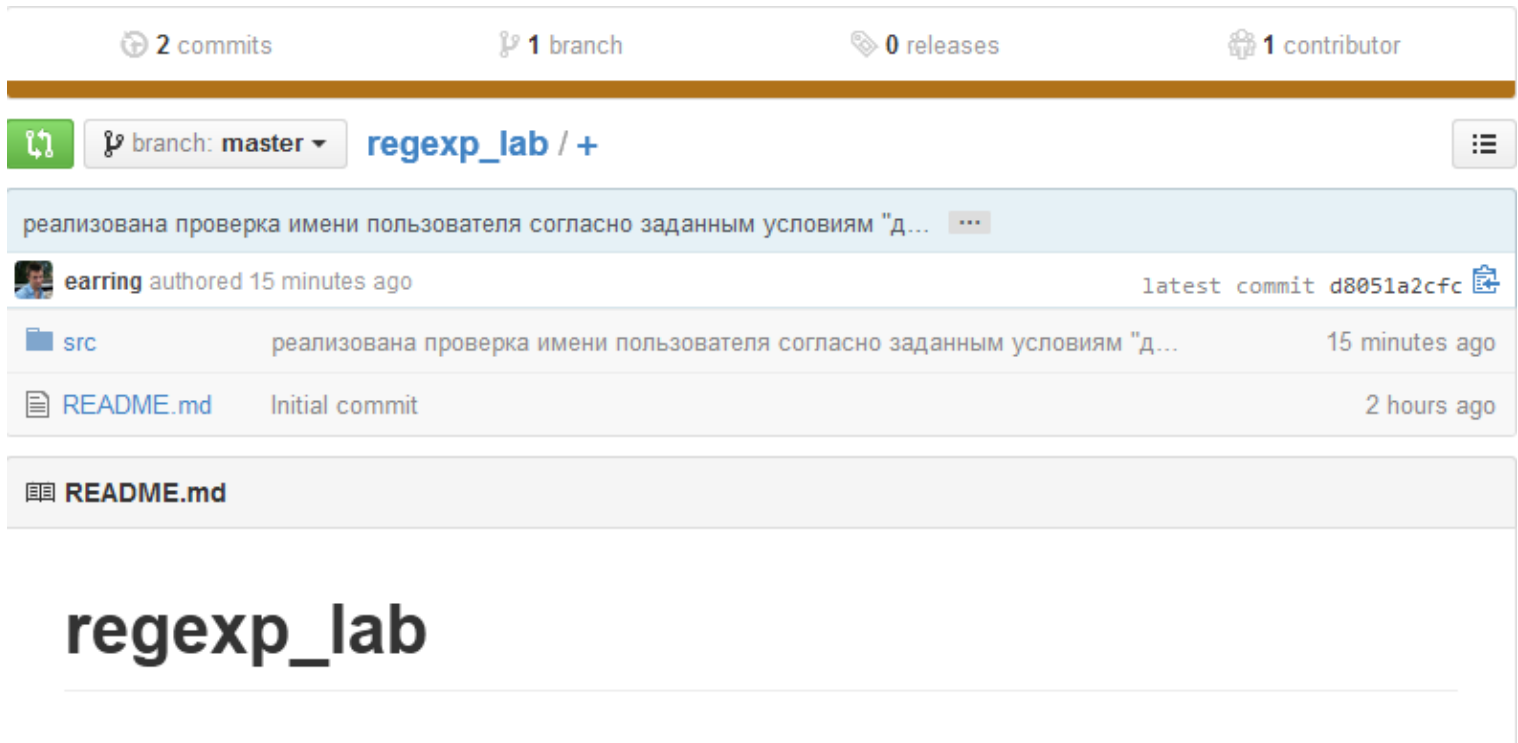


После выбора “Push” коммиты будут отправлены на сервер. Обзор изменений теперь выглядит следующим образом.



Можно увидеть, что метка “origin/master” теперь находится на последнем коммите, а не на самом первом, это значит, что теперь в удаленном репозитории (они по умолчанию называются origin) последним коммитом является тот же коммит, что является последним в локальном репозитории. Иными словами, набор коммитов на сервере и локальном компьютере стал одинаковым. Однако это тесно связано с понятием ветвления в Git, которое будет рассмотрено в следующей лабораторной работе.

Можно также увидеть, что в репозитории в GitHub файлы также изменились.



Внесем некоторые изменения в код программы.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class UserNameChecker {

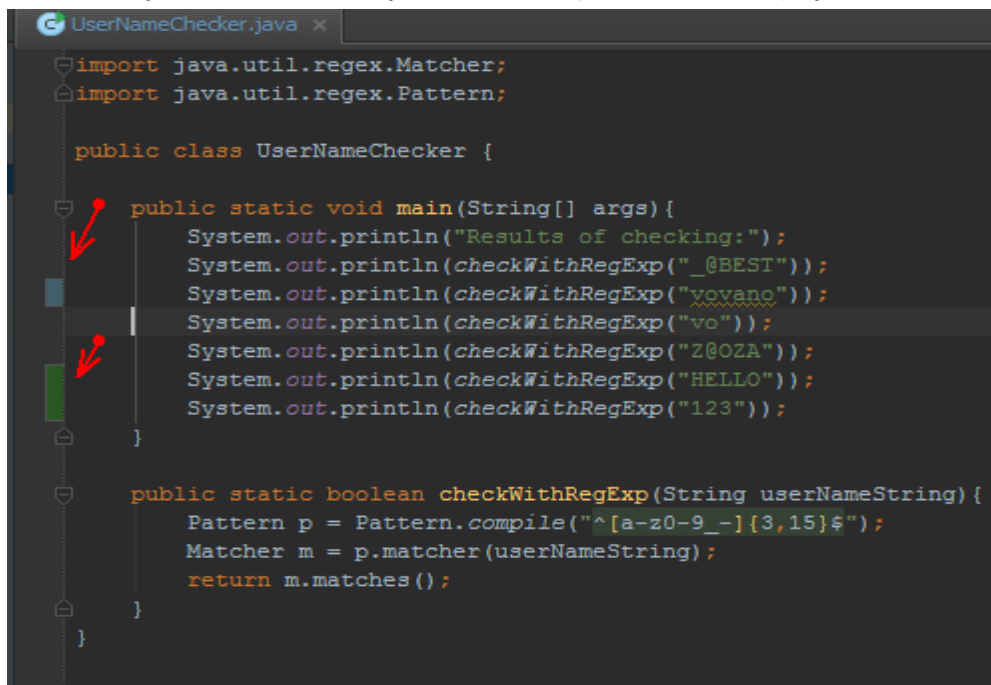
    public static void main(String[] args){
        System.out.println("Results of checking:");
        System.out.println(checkWithRegExp("_@BEST"));
        System.out.println(checkWithRegExp("vovano"));
        System.out.println(checkWithRegExp("vo"));
        System.out.println(checkWithRegExp("Z@OZA"));
        System.out.println(checkWithRegExp("HELLO"));
        System.out.println(checkWithRegExp("123"));
    }
}
```

```

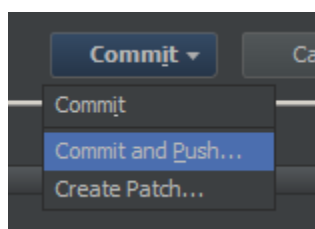
public static boolean checkWithRegExp(String userNameString) {
    Pattern p = Pattern.compile("[a-z0-9_-]{3,15}$");
    Matcher m = p.matcher(userNameString);
    return m.matches();
}
}

```

Обратите внимание на цветные полосы возле строк кода. Синий цвет означает изменившуюся строку, зеленый - добавившуюся, а на месте удаленных строк появится треугольник.



Для удобства процесс отправки коммита в локальный репозиторий и последующий процесс отправки всех коммитов в удаленный репозиторий можно выполнить одной командой. Необходимо перейти в меню “Commit Changes”, только вместо команды “Commit” выполнить команду “Commit and Push”.



Далее нужно провести процедуру push как сказано выше.

В процессе изменений программы процессы commit и push повторяются. Однако это самый простой вариант работы, когда над программой работает только один человек. При работе нескольких человек приходится обеспечивать их взаимодействие, в том числе, используя ветвление. Это будет рассмотрено подробно на следующей лабораторной работе.

Пример, использовавшийся в практической части, доступен по адресу https://github.com/earring/regexp_lab.

Задания для самостоятельной работы

Программа выполняется **индивидуально**. Необходимо реализовать программу своего варианта, используя Git и размещая её репозиторий на GitHub. Программа должна реализовываться поэтапно, т. е. каждое значительное изменение кода оформляется в виде коммита с ясным комментарием. Рекомендуется продемонстрировать использование регулярных выражений.

Вариант №1, 7, 13, 19, 25

В программу передается текст из файла. В этом тексте должно быть подсчитано количество прилагательных, наречий и глаголов.

Вариант №2, 8, 14, 20, 26

В программу подается текст из файла, состоящего из строчек формата

Имя Фамилия | Возраст | ТелефонныйНомер | ЭлектроннаяПочта

Необходимо проверить данные на корректность, и по возможности, исправленную версию поместить в другой файл. Если данные ошибочные, то часть строки оставить пустой. К примеру, из строки

ИванИванов|-27|+7999000 1 1 11|example@@yandex..ru

может после исправления получиться строка

Иван Иванов|27|+7 (999) 000-11-11|example@yandex.ru

Вариант №3, 9, 15, 21, 27

В программу подается текст из файла, в котором находится секретное письмо. Его надо очистить от конфиденциальных данных. Конфиденциальными данными считаются имена, фамилии, номера телефонов и данные, помогающие определить географическое положение адресата. Вместо данных, которые программа находит конфиденциальными, следует написать [censored]. Рекомендуется проверять работоспособность программы на художественных текстах.

Вариант №4, 10, 16, 22, 28

В программу подается текст из файла, в котором находится некий текст, либо набор текстов. Необходимо определить “настроение” текста. Это значит, что если в тексте будет употреблено много негативных слов, то текст будет иметь отрицательное значение “настроения”. Должно быть вычислено точно число “настроения”, для возможности сравнения различных текстов.

Вариант №5, 11, 17, 23, 29

В программу подается текст из файла, в котором находится текст, в котором нужно заменить числа, написанные прописью, на числа, написанные цифрами. Например, текст “сто одиннадцать тысяч фиолетовых оленей” будет записан как “111 000 фиолетовых оленей”. Считать, что числа прописью написаны без ошибок в грамматике, падеже и т. д.

Вариант №6, 12, 18, 24, 30

В программу передается текст из файла, в котором находится текст, в котором нужно во всех найденных телефонных номерах изменить код страны, а также привести номера к единому формату. К примеру, текст “Звоните по номеру +79000000000” должен быть заменен на текст “Звоните по номеру +1 (900) 000-00-00”. Возможные ошибки в написании телефонного номера учитывать.

Литература, ссылки

1. <http://git-scm.com/book/ru/v1>
2. <http://www.quizful.net/post/Java-RegExp>
3. <http://www.friendlyfunction.com/ru/using-regular-expressions-java/>