

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**

**РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ В.Ф. УТКИНА**

Утилиты разработки программного обеспечения

Методические указания
к практическим работам



Рязань 2020

УДК 004.4'23

Утилиты разработки программного обеспечения:
методические указания к лабораторным работам / Рязан. гос.
радиотехн. ун-т; сост.: А.И. Ефимов. – Рязань, 2020. – 32 с.

Содержат указания по выполнению лабораторных работ для студентов, обучающихся по направлениям «Математическое обеспечение и администрирование информационных систем», «Информатика и вычислительная техника», «Бизнес-информатика» уровня бакалавриата.

Предназначены для бакалавров дневного, вечернего и заочного отделений.

Ил. 15. табл. 1.

*Системы контроля версий, системы отслеживания ошибок,
Subversion, Git, автоматизированное тестирование*

Печатается по решению редакционно-издательского совета
Рязанского государственного радиотехнического университета.

Рецензент: кафедра электронных вычислительных машин
Рязанского государственного радиотехнического университета
имени В.Ф. Уткина (зав. кафедрой Б.В. Костров)

Инструментальные средства разработки
программного обеспечения

Составители: Ефимов Алексей Игоревич

Рязанский государственный радиотехнический университет
имени В.Ф. Уткина.

390005, Рязань, ул. Гагарина, 59/1.

Редакционно-издательский центр РГРТУ.

Лабораторная работа № 1. Subversion. Основные операции.

Цель работы: Получение навыков работы с системой контроля версий Subversion (хранилищем и рабочими копиями).

Теоретическая часть

1 Хранилище

Subversion является централизованной системой для хранения информации. Ее основа – хранилище, являющееся центром хранения данных. Оно хранит информацию в форме дерева файлов.

Любое количество клиентов может подключиться к хранилищу и прочитать или записать новые версии этих файлов. Записывая данные, клиент делает информацию доступной для остальных; читая данные, клиент получает информацию от других.

Subversion запоминает каждое внесённое изменение: любое изменение любого файла, а также изменения в самом дереве каталогов, такие как добавление, удаление и реорганизация файлов и каталогов. При чтении данных из хранилища клиент обычно видит как последнюю версию дерева файлов, так и предыдущие состояния файловой системы.

2 Правки

При фиксации версии создаётся новое состояние файловой системы – правка (ревизия). Каждая правка получает уникальный номер, на единицу больший номера предыдущей. Начальная правка только что созданного хранилища получает номер 0 и не содержит ничего, кроме пустого корневого каталога.

В служебном каталоге `.svn/` для каждого файла рабочего каталога Subversion записывает информацию о двух важнейших свойствах:

- на какой правке основан рабочий файл (рабочая правка файла);
- временной метке, определяющей, когда рабочая копия последний раз обновлялась из хранилища.

Используя эту информацию при соединении с хранилищем, Subversion может сказать, в каком из следующих четырех состояний находится рабочий файл:

1) Файл не изменялся и не устарел.

Файл не изменялся в рабочем каталоге, в хранилище не фиксировались изменения этого файла со времени создания его рабочей правки. Команды `svn commit` и `svn update` никаких операций производить не будут;

2) Файл изменялся локально и не устарел.

Файл был изменен в рабочей копии, но в хранилище не фиксировались его изменения. Есть локальные изменения, которые не

были зафиксированы в хранилище, поэтому `svn commit` выполнит фиксацию ваших изменений, а `svn update` не сделает ничего;

3) Файл не изменялся и устарел.

В рабочем каталоге файл не изменялся, но был изменен в хранилище. Необходимо выполнить обновление файла для того, чтобы он соответствовал текущей правке. Команда `svn commit` не сделает ничего, а `svn update` обновит рабочую копию файла;

4) Файл изменялся локально и устарел.

Файл был изменен как в рабочем каталоге, так и в хранилище. Команда `svn commit` выдаст ошибку «out-of-date». Файл необходимо сначала обновить; `svn update` попытается объединить локальные изменения с опубликованными. Если Subversion не сможет совершить объединение, она предложит пользователю разрешить конфликт вручную.

Фундаментальные правила Subversion – «передающее» действие не приводит к «принимаемому», и наоборот. То, что вы готовы внести изменения в хранилище, не означает, что вы готовы принять изменения от других. А если вы все еще работаете над новыми изменениями, то `svn update` объединит изменения из хранилища с вашими собственными вместо того, чтобы заставить вас опубликовать их.

3 Простейший рабочий цикл

Типичный рабочий цикл с применением Subversion выглядит примерно так:

1) обновление рабочей копии:

svn update

2) внесение изменений:

svn add (delete, copy, move)

3) редактирование файлов под контролем Subversion;

4) анализ изменений:

svn status (diff, revert)

5) слияние изменений, выполненных другими, с вашей рабочей копией:

svn update

svn resolved

6) фиксация изменений:

svn commit

3.1 Создание репозитория

Для создания пустого репозитория необходимо выполнить команду `svnadmin create` (команда `svnadmin` поставляется со стандартным пакетом `svn`):

Пример 1. Создание пустого репозитория:

```
$ svnadmin create /path/to/rep
```

Здесь /path/to/rep – это URL того адреса, по которому будет создан репозиторий.

Для импортирования нового проекта в Subversion-хранилище используется `svn import`.

Команда `svn import` это быстрый способ скопировать не версионированное дерево файлов в хранилище, создавая при необходимости промежуточные директории.

Пример 2. Создание пустого репозитория в подкаталоге /usr/local/svn/newrepos, затем перенесение всего дерева каталогов в этот репозиторий (`file:///usr/local/svn/newrepos/some/project`):

```
$ svnadmin create /usr/local/svn/newrepos
```

```
$ svn import mytree file:///usr/local/svn/newrepos/some/project \  
-m "Initial import"
```

```
Adding mytree/foo.c
```

```
Adding mytree/bar.c
```

```
Adding mytree/subdir
```

```
Adding mytree/subdir/quux.h
```

```
Committed revision 1.
```

В предыдущем примере выполняется копирование содержимого директории `mytree` в директорию `some/project` хранилища.

3.2 Создание рабочей копии репозитория

Для создания рабочей копии используется команда `svn checkout`

Для того чтобы создать рабочую копию уже существующего репозитория, вам нужно получить какую-либо из подкаталогов хранилища.

Рабочая копия представляет собой обычное дерево каталогов на компьютере.

После изменения файлов рабочей копии нужно «опубликовать» изменения, в результате чего они станут доступными для всех участников проекта.

Рабочая копия содержит дополнительные файлы в подкаталоге `.svn`. Они помогают определить файлы рабочей копии, содержащие неопубликованные изменения, и файлы, устаревшие по отношению к файлам других участников.

Пример 3. Создание рабочей копии репозитория `http://svn.example.com/repos/calc`, загружается в текущую директорию:

```
$ svn checkout http://svn.example.com/repos/calc
```

```
A calc/Makefile
```

```
A calc/integer.c
```

A calc/button.c

Checked out revision 56.

3.3 Внесение изменений в рабочую копию

Изменения, которые можно сделать в рабочей копии:

- изменения файлов;
- изменения в структуре каталогов.

Подкоманды, наиболее часто используемые при внесении изменений:

- `svn add <file>` – запланировать для добавления в хранилище. При фиксации `<file>` станет компонентом своей родительской директории;
- `svn delete <file>` – запланировать удаление из хранилища;
- `svn copy <file1> <file2>` – создать элемент `<file2>` как копию `<file1>`;
- `svn move <file1> <file2>` – `<file2>` будет запланирован для добавления как копия `<file1>`, а `<file1>` будет запланирован для удаления.

3.4 Фиксация (публикация) изменений в хранилище

Публикация (коммит) представляет собой «фиксирование» данных в репозитории. Для публикации изменений следует воспользоваться командой `commit`. Можно использовать ключ `-m` для указания поясняющего сообщения в одной строке:

Пример 4. Фиксация изменений с сообщением «First message»:

```
$ svn commit -m "First message."
```

```
Sending
```

```
button.c
```

```
Transmitting file data
```

```
Committed revision 2.
```

3.5. Просмотр истории изменений

`Svn log` показывает вам развернутую информацию: лог сообщения, присоединенные к правкам, с указанной датой изменений и их автором, а также измененные в правке пути файлов.

Пример 5. Команда `svn log`:

```
$ svn log
```

```
-----  
r3 | sally | Mon, 15 Jul 2002 18:03:46 -0500 | 1 line  
Added include lines and corrected # of cheese slices.
```

```
-----  
r2 | harry | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line  
Added main() methods.
```

```
-----  
r1 | sally | Mon, 15 Jul 2002 17:40:08 -0500 | 1 line  
Initial import
```

Обратите внимание на то, что по умолчанию лог сообщения выводятся в обратном хронологическом порядке. При необходимости порядок вывода информации об истории изменений можно изменить.

3.5. Другие полезные команды

`svn help`

Клиент для командной строки Subversion является самодокументируемым — в любой момент команда `svn help <subcommand>` покажет описание синтаксиса, параметров и поведения подкоманды `subcommand`.

`svn info`

Позволяет просматривать информацию о репозитории, такую как URL родительского репозитория, номер текущей ревизии, дату последних изменений.

`svn status`

Служит для того, чтобы узнать состояние любого элемента в вашей рабочей копии. Можно использовать перед фиксацией для просмотра изменений, которые войдут в следующую фиксацию.

Метки

Можно считать, что метки — это специальные обозначения, прикреплённые к коммиту. С помощью них удобно искать коммиты, в которых приложение было в определённом состоянии (например, релиз 1.0, релиз 1.2 и т.д.).

Для меток в Subversion нет специальной команды, они выглядят как дешёвые копии (cheap copies), или ссылки:

Пример 6. Создание метки 1.0 в директории `tags` и сообщением "Release 1.0":

```
svn copy http://svn.example.com/project/trunk \
http://svn.example.com/project/tags/1.0 -m "Release 1.0"
```

Практическая часть

Используя клиент Subversion необходимо проделать, задокументировать и отразить в отчете следующие задания:

1. Создать репозиторий в любой выбранной пустой директории.
2. Создать рабочую копию в любой выбранной пустой директории.
3. Проверить, на какое хранилище ссылается созданная рабочая копия.
4. Просмотреть последнюю дату изменения файлов в репозитории.
5. Создать три текстовых файла (`f1.txt`, `f2.txt`, `f3.txt`) с несколькими содержательными строками внутри каждого из них.

6. Добавить в отслеживаемые все созданные файлы.
7. Просмотреть перед фиксацией сделанные изменения.
8. Зафиксировать изменения с любым осмысленным сообщением.
9. Удалить файл f3.txt и зафиксировать его удаление с любым осмысленным сообщением.
10. Посмотреть номер текущей ревизии.
11. Посмотреть историю коммитов, задокументировать.
12. Совершить еще 2 изменения и коммита, поставить на одном из них метку «Лабораторная_1», просмотреть полученные результаты.

Содержание отчёта

По результатам выполнения работы оформляется отчет в соответствии с требованиями ГОСТ 7.32-2017 «Отчет о научно-исследовательской работе. Структура и правила оформления», включающий:

- титульный лист;
- цель работы;
- описание структуры хранилища во время выполнения (при выполнении операций, меняющих состояние хранилища);
- выполняемые команды с комментариями и результаты их выполнения;
- выводы.

Контрольные вопросы:

1. Опишите жизненный цикл коммитов в Svn.
2. Что представляет из себя коммит?
3. Как хранятся изменения между различными файлами в Svn?
4. Зачем создается каталог .svn в каждой директории?
5. Перечислите известные вам достоинства и недостатки Subversion.
6. После операции svn update имеем A file1. Поясните значение данного статуса.
7. Откуда берется копия файла для замены при осуществлении команды svn revert?
8. Опишите быстрый способ скопировать неверсионированное дерево в хранилище.
9. Действия, вызванные svn commit и svn update, при условии, что файл изменялся локально и устарел.
10. Перечислите основные команды subversion для внесения изменений в рабочую копию.

11. По каким протоколам можно получить доступ к хранилищу Subversion?

12. В чём отличие модели «блокирование – изменение – разблокирование» от «копирование – изменение – слияние»?

13. В каком порядке выводится история коммитов при вызове команды `svn log`?

14. Как в Subversion обозначается последняя (самая новая) правка хранилища, номер правки элемента рабочей копии, правка, в которой элемент последний раз редактировался, и правка, предшествующая последней правке?

15. Какие два типа игнорирования файлов существуют в Subversion? Чем они отличаются?

16. Опишите назначение игнорирования файлов?

Лабораторная работа № 2. Subversion. Ветвления.

Цель работы

Получение навыков работы с системой контроля версий Subversion (ветками и разрешением конфликтов).

Теоретическая часть

1. Ветвление

Ветка — это направления разработки, которое существует независимо от другого направления, однако имеющие с ним общую историю, если заглянуть немного в прошлое. Ветка всегда берет начало как копия чего-либо и движется от этого момента, создавая свою собственную историю.

Создание ветки

Создать ветку очень просто — при помощи команды `svn copy` делаете в хранилище копию проекта.

Обратите внимание, что директория для ветви уже должна существовать в репозитории.

Пример 1. Создание директорий для ветвей (в исходном репозитории, не в копии):

```
$ mkdir -p branches/test-branch
$ svn commit -m "folders for branching"
Adding      branches
Adding      branches/test-branch
Committing transaction...
Committed revision 3.
```

Команда `svn copy` может оперировать с двумя URL напрямую.

Пример 2. Создание ветки «test-branch» и фиксирование этого изменения:

```
svn          copy          file:///home/uname/repository/trunk
file:///home/uname/repository/branches/test-branch -m "first branch"
Committing transaction...
Committed revision 4.
```

Эта процедура проста в использовании, так как нет необходимости в создании рабочей копии, отражающей большое хранилище. Вам вовсе можно не иметь рабочей копии.

Работа с веткой

После создания ветки проекта, можно создать новую рабочую копию для начала ее использования:

Пример 3. Создание рабочей копии ветки «test-branch»:

```
$ svn checkout file:///home/uname/repository/branches/test-branch
A test-branch/Makefile
A test-branch/integer.c
```

Checked out revision 5.

Ключевые идеи, стоящие за ветками:

В отличие от других систем управления версиями, ветки в Subversion существуют в хранилище не в отдельном измерении, а как обычные нормальные директории файловой системы. Такие директории просто содержат дополнительную информацию о своей истории.

Subversion не имеет такого понятия как ветка — есть только копии. При копировании директории результирующая директория становится «веткой» только потому, что вы рассматриваете ее таким образом. Вы можете по-разному думать о директории, по-разному ее трактовать, но для Subversion это просто обычная директория, которая была создана копированием.

2. Копирование изменений между ветками

В проектах, имеющих большое количество участников, когда кому-то необходимо сделать долгосрочные изменения, которые возможно нарушат главную линию, стандартной процедурой является создание отдельной ветки и фиксация изменения туда до тех пор, пока работа не будет полностью завершена.

Чтобы избежать слишком большого расхождения веток, можно продолжать делиться изменениями по ходу работы. А тогда, когда ваша ветка будет полностью закончена, полный набор изменений ветки может быть скопирован обратно в основную ветку.

Пример 4. Копирование изменений между ветками: из b1 в b2:

```
svn merge url://to/branch/b2 url://to/branch/b1
```

3. Конфликты при объединении

Команда `svn merge` иногда не может гарантировать корректного поведения: пользователь может запросить сервер сравнить любые два дерева файлов, даже такие, которые не имеют отношения к рабочей копии! Из этого следует большое количество потенциальных человеческих ошибок.

Команда `svn merge` в конфликтной ситуации создает три файла с названиями `filename.working`, `filename.left` и `filename.right`. Здесь, термины «left» и «right» указывают на две стороны сравнения, то есть на используемые при сравнении деревья.

Способы разрешения конфликтов

Предположим что вы запустили `svn update` и получили:

Пример 5. Конфликтный файл при объединении изменений:

```
$ svn update
U INSTALL
G README
```

C bar.c

Updated to revision 46.

Файл, отмеченный флагом С, имеет конфликт. Это значит, что изменения с сервера пересеклись с вашими личными, и теперь вам нужно вручную сделать между ними выбор.

Всякий раз, когда возникает конфликт, для того, чтобы помочь вам заметить и решить этот конфликт, происходят, как правило, три вещи:

1) Subversion печатает флаг С во время обновления и запоминает, что файл в состоянии конфликта;

2) Если Subversion считает, что файл объединяемого типа, она помещает маркеры конфликта — специальные текстовые строки которые отделяют «стороны» конфликта — в файл, для того, чтобы визуально показать пересекающиеся области;

3) Для каждого конфликтного файла Subversion добавляет в рабочую копию до трех не версионированных дополнительных файлов:

- filename.mine (ваш файл в том виде, в каком он был в рабочей копии до обновления (если Subversion решает, что файл не объединяемый, тогда файл .mine не создается, так как он будет идентичным рабочему файлу));

- filename.OLDREV (файл правки BASE, где BASE – правка которая была до обновления рабочей копии, то есть это файл, который у вас был до внесения изменений);

- filename.NEWREV (файл, который ваш Subversion-клиент только что получил с сервера, когда вы обновили рабочую копию. Этот файл соответствует правке HEAD хранилища, где HEAD — специальный указатель на последний коммит в хранилище).

Здесь OLDREV - это номер правки файла в директории .svn, а NEWREV - это номер правки HEAD хранилища.

Если вы получили конфликт, у вас есть три варианта:

- Объединить конфликтующий текст «вручную» (путем анализа и редактирования маркеров конфликта в файле);

- Скопировать один из временных файлов поверх своего рабочего файла;

- Выполнить *svn revert <filename>* для того, чтобы убрать все ваши локальные изменения.

После того, как вы решили конфликт, вам нужно поставить в известность Subversion, выполнив *svn resolved*. Эта команда удалит три временных файла, и Subversion больше не будет считать, что файл находится в состоянии конфликта.

3.1. Объединение конфликтов вручную

Конфликтный файл будет выглядеть так:

```
<<<<<<< имя файла
```

```
ваши изменения
```

```
=====
```

```
результат автоматического слияния с репозиторием
```

```
>>>>>>> ревизия
```

Вам необходимо разрешить конфликты вручную редактированием данного файла или через сторонние приложения.

Когда вы произведете слияние изменений, выполните *svn resolved*, то вы готовы к фиксации изменений:

Пример 6. Фиксирование состояния разрешения конфликта для файла *sandwich.txt*:

```
$ svn resolved sandwich.txt
```

И, наконец, зафиксируйте изменения:

```
$ svn commit -m "Go ahead and use my sandwich, discarding Sally's edits."
```

3.2. Использование *svn revert*

Если вы получили конфликт и вместо анализа решаете отбросить локальные изменения и начать сначала, просто отмените их:

Пример 7. Отмена изменений в файле *sandwich.txt*:

```
$ svn revert sandwich.txt
```

```
Reverted 'sandwich.txt'
```

Обратите внимание, что когда вы возвращаете файл к предыдущему состоянию, вам не нужно выполнять команду *svn resolved*.

Практическая часть

Замечание: забирайте необходимые изменения у напарника и отправляйте их в репозиторий, когда это необходимо.

1. Создайте пустой проект в любой директории.
2. Создайте в ветви *trunk* один файл с 15-20 строками программного кода.
3. Создайте ветви с вашим именем.
4. Склонируйте к себе локальную копию вашей ветви без клонирования всего проекта.
5. Отредактируйте созданный файл так, чтобы номера изменяемых строк у вас и напарника совпадали. Зафиксируйте изменения в ваших ветках.
6. Влейте изменения ветки напарника к себе в ветку. Отрадите в сообщении к коммиту, то, что это коммит с вливанием веток.
7. Разрешите возникшие конфликты.
8. Опишите ситуации, когда конфликт происходит, а когда нет.
9. Влейте изменения в ветку *trunk*.

10. Сделайте ещё два любых коммита в своих ветках.
11. Отмените последний коммит.

Содержание отчёта

По результатам выполнения работы оформляется отчет в соответствии с требованиями ГОСТ 7.32-2017 «Отчет о научно-исследовательской работе. Структура и правила оформления», включающий:

- титульный лист;
- цель работы;
- описание структуры хранилища во время выполнения (при выполнении операций, меняющих состояние хранилища);
- выполняемые команды с комментариями и результаты их выполнения;
- выводы.

Контрольные вопросы:

1. Опишите, в чем состоит отличие локального репозитория от удалённого.
2. Назовите способы копирования проекта к себе в удалённый репозиторий.
3. Как отредактировать сообщение у коммита?
4. Что такое ветка?
5. Что представляет собой ветка в Subversion?
6. Перечислите недостатки работы в одной ветке.
7. Как скопировать отдельные изменения между ветками?
8. Как скопировать изменения из одной ветки в другую?
9. Какие файлы создаются при конфликтной ситуации при слиянии?
10. Опишите известные вам способы разрешения конфликтов в Subversion и их отличительные особенности.

Лабораторная работа № 3. Git. Основные операции.

Цель работы

Получение навыков работы с системой контроля версий Git (хранилищем и рабочими копиями).

Продолжительность работы: 2 часа.

Теоретическая часть

1. Хранилище

Git — распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано.

Git является распределённой системой контроля версий. В Git используется своя собственная модель разработки, основанная на «снимках» системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков**.

Git-директория — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера.

Рабочая директория является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git-директории и располагаются на диске, для того чтобы их можно было изменять и использовать.

Область подготовленных файлов (staging area) — это файл, располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют «индекс», или stage-область.

2. Первоначальная настройка репозитория

Когда вы установили Git, необходимо настроить репозиторий, чтобы можно было различить именно Вас при фиксировании изменений.

Ваши настройки могут храниться в следующих файлах:

`/etc/gitconfig` - содержит значения, общие для всех пользователей системы и для всех их репозиториях;

`~/.gitconfig` или `~/.config/git/config` хранит настройки конкретного пользователя. Этот файл используется при указании параметра `-global`;

Файл `config` в каталоге Git'a (т.е. `.git/config`) в том репозитории, который вы используете, хранит локальные настройки.

Имя пользователя и почта

Указание вашего имени и адреса электронной почты критически важно, потому что каждый коммит в Git'е содержит эту информацию. Она включена в коммиты, передаваемые вами, и не может быть далее изменена:

Пример 1. Конфигурирование имени пользователя и электронной почты через консоль:

```
$ git config --global user.name "Your Name"
$ git config --global user.email your_name@example.com
```

Проверка настроек

Для проверки используемой конфигурации, можно использовать команду **git config --list**, чтобы показать все настройки, которые найдёт Git:

Пример 2. Показать все настройки конфигурации:

```
$ git config --list
user.name=Your Name
user.email=your_name@example.com
```

...

Также вы можете проверить значение конкретного ключа, выполнив `git config <key>`:

Пример 3. Отображение заданной настройки конфигурации (user.name):

```
$ git config user.name
Your Name
```

3. Простейший рабочий цикл в Git

Типичный рабочий цикл выглядит примерно так:

1) обновление рабочей копии:

```
git pull;
```

2) внесение изменений обычными командами для удаления, создания файлов;

3) редактирование файлов под контролем git;

4) анализ изменений:

```
git status (diff, revert);
```

5) слияние изменений, выполненных другими, с вашей рабочей копией:

```
git pull
```

+ редактирование конфликтов (в случае необходимости)

```
git commit;
```

6) фиксация изменений:

```
git commit;
```

3.1. Создание репозитория

Для создания нового проекта в текущей директории, необходимо выполнить:

```
$ git init
```


Эта команда создаёт в текущей директории новую поддиректорию с именем `.git`, содержащую все необходимые файлы репозитория — основу Git-репозитория.

В этой директории уже могут содержаться файлы проекта, так что этот способ можно рассматривать и как создание репозитория из существующего исходного кода.

Пример 4. Создание пустого Git-репозитория:

```
$ git init
Initialized empty Git repository in
/home/tkseniya/University/newDir/.git/
```

3.2. Клонирование репозитория

Так же проект может уже существовать и располагаться на удалённом сервере. В этом случае получить его копию можно, используя команду **git clone <url>**.

При выполнении `git clone` с сервера забирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любую из копий на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования.

Пример 5. Клонирование репозитория `libgit2` из `github`:

```
$ git clone https://github.com/libgit2/libgit2
Cloning into 'libgit2'...
remote: Enumerating objects: 88418, done.
remote: Counting objects: 100% (88418/88418), done.
remote: Compressing objects: 100% (24731/24731), done.
remote: Total 88418 (delta 61958), reused 88414 (delta 61956),
pack-reused 0
Receiving objects: 100% (88418/88418), 40.19 MiB | 627.00 KiB/s,
done.
Resolving deltas: 100% (61958/61958), done.
Checking out files: 100% (6029/6029), done.
```

В Git реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол `https://`, вы также можете встретить `git://` или `user@server:path/to/repo.git`, использующий протокол передачи SSH.

3.3. Внесение изменений в рабочую копию

Изменения, которые можно сделать в рабочей копии:

- изменения файлов;
- изменения в структуре.

Подкоманды, наиболее часто используемые при внесении изменений:

- `git add <file>` - запланировать для добавления в хранилище.

При фиксации `<file>` станет компонентом своей родительской директории;

- `git add *` - добавление всех файлов в хранилище (кроме игнорируемых).

3.4. Просмотр истории изменений

Одним из основных и наиболее мощных инструментов для просмотра истории коммитов является команда **git log**.

Пример 6. Вывод команды `git log`:

```
$ 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 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` перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядку. Последние коммиты находятся вверху. Из примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 контрольными суммами, именем и электронной почтой автора, датой создания и сообщением коммита.

Одним из самых полезных аргументов является `-p`, который показывает разницу, внесенную в каждый коммит. Так же вы можете использовать аргумент `-n`, который позволяет установить лимит на вывод количества коммитов (в количестве `n`).

Опция **--pretty=format** отображает лог в удобном для чтения виде. С параметром и этой опции вы можете ознакомиться в соответствующем разделе справки, а здесь приведём пример:

Пример 7. Вывод лога коммитов в красивой форме:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
/\
/ * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | e1193f8 support for heads with slashes in them
//
* d6016bc require time for xmlschema
```

где `%h` — сокращённый хэш коммита, а `%s` — сообщение коммита.

3.5. Метки

Как и большинство СКВ, Git имеет возможность пометить (тегировать, tag) определённые моменты в истории как важные. Как правило, эта функциональность используется для отметки моментов выпуска версий (v1.0, и так далее).

3.5.1. Просмотр меток

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

Пример 8. Просмотр имеющихся меток в репозитории:

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

Данная команда перечисляет метки в алфавитном порядке.

3.5.2. Создание меток

Git использует два основных типа меток: легковесные и аннотированные.

Легковесная метка — это указатель на определённый коммит.

Аннотированные метки хранятся в базе данных Git'а как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий. Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.

По умолчанию, команда git push не отправляет метки на удалённые сервера.

Аннотированные метки

Для создания аннотированной метки укажите -a при выполнении команды tag (сообщение добавляется с ключом -m):

Пример 9. Создание аннотированной метки с сообщением «my version 1.4» и меткой v1.4:

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

Вы можете посмотреть данные метки вместе с коммитом, который был помечен, с помощью команды git show:

Пример 10. Просмотр метки с тегом «v1.4»:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700
```

```
my version 1.4
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

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 передавайте опций -a, -s и -m:

Пример 11. Создание легковесной метки с тегом «v1.4-lw»:

```
$ git tag v1.4-lw
```

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

Пример 12. Показ легковесной ветки с тегом «v1.4-lw»:

```
$ git show v1.4-lw
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

changed the version number

Выставление меток позже

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

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch
'experiment'
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
```

Для отметки коммита укажите его контрольную сумму (или её часть) в конце команды:

Пример 15. Выставлении метки с тегом «v1.2» на коммит с хэш-суммой 9fceb02 позже:

```
$ git tag -a v1.2 9fceb02
```

3.6. Другие полезные команды

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

```
$ git help <глагол>
```

```
$ git <глагол> --help
```

```
$ man git-<глагол>
```

Например, так можно открыть руководство по команде `config`

```
$ git help config
```

Практическая часть

I. Настройка глобального рабочего пространства

1. Задать в конфигурационном файле имя пользователя.
2. Задать в конфигурационном файле электронную почту.
3. Просмотреть сделанные настройки двумя способами.

II. Основные операции

1. Создать локальный репозиторий.

2. Создать три текстовых файла (f1.txt, f2.txt, f3.txt) с несколькими содержательными строками внутри.
3. Добавить в отслеживаемые файлы f1.txt, f3.txt.
4. Зафиксировать изменения с любым осмысленным сообщением.
5. Удалить f3.txt
6. Зафиксировать изменения с любым осмысленным сообщением.
7. Посмотреть историю коммитов, задокументировать.
8. Сделать ещё два коммита и поставить тег (любым способом) «Лабораторная_3», просмотреть результат: тег сообщения с коммитом.

Содержание отчёта

По результатам выполнения работы оформляется отчет в соответствии с требованиями ГОСТ 7.32-2017 «Отчет о научно-исследовательской работе. Структура и правила оформления», включающий:

- титульный лист;
- цель работы;
- описание структуры хранилища во время выполнения (при выполнении операций, меняющих состояние хранилища);
- выполняемые команды с комментариями и результаты их выполнения;
- выводы.

Контрольные вопросы:

1. Кто является создателем системы контроля версий Git?
2. Опишите жизненный цикл коммитов в Git.
3. Что представляет из себя коммит?
4. Что такое хэш-сумма? Как коммит соотносится с хэш-суммой?
5. Какой алгоритм хэширования использует Git?
6. Что представляет из себя «слепок» файловой системы?
7. Как выглядит жизненный цикл одного коммита в Git? Объясните понятия рабочая директория (working directory), область подготовленных файлов (staging area).
8. Чем отличается глобальное и локальное конфигурирование параметров?
9. Зачем создаётся каталог .git в каждой директории?
10. Перечислите известные вам достоинства Git.
11. Что происходит при командах git add file.txt?
12. Опишите быстрый способ скопировать неверсионированное дерево в хранилище.

13. Перечислите основные команды Git для внесения изменений в рабочую копию.
14. Какие бывают метки в Git?
15. По каким протоколам можно получить доступ к хранилищу Git?
16. В каком порядке выводится история коммитов при вызове команды «git log»?
17. Как в Git обозначается последняя (самая новая) правка хранилища, как хранится?
18. Зачем нужно игнорирование файлов?

Лабораторная работа № 4. Git. Совместная работа.

Цель работы

Получение навыков работы с системой контроля версий Git, используя GitLab (ветви, разрешение конфликтов).

1. Теоретическая часть

1.1. Ветвление

Ветка — это направления разработки, которое существует независимо от другого направления, однако имеющие с ним общую историю, если заглянуть немного в прошлое. Ветка всегда берет начало как копия чего-либо и движется от этого момента создавая свою собственную историю.

Ветвление Git очень легковесно. Операция создания ветки выполняется почти мгновенно, переключение между ветками - также быстро. В отличии от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день.

Ветка (branch) в Git — это легко перемещаемый указатель на один из коммитов. Имя основной ветки по умолчанию в Git — master.

Когда вы делаете коммиты, то получаете основную ветку, указывающую на ваш последний коммит. Каждый коммит автоматически двигает этот указатель вперед.

Что же на самом деле происходит, когда вы создаете ветку? Всего лишь создается новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем “testing” Вы можете это сделать командой **git branch** :

Пример 1. Создание ветки testing. При создании мы останемся на той ветке, на которой были:

```
$ git branch testing
```

Чтобы переключиться на существующую ветку, выполните команду **git checkout**. Давайте переключимся на ветку “testing”:

Пример 2. Перемещение на ветку testing:

```
$ git checkout testing
```

Слияние веток

Слияние веток выполняется с помощью команды **git merge <branchname>**. Например, если мы хотим влить изменения из ветки iss53 в текущую ветку, в которой мы находимся, необходимо выполнить:

Пример 3. Вливание изменений из ветки iss53 в текущую ветку (на которой мы сейчас находимся):

```
$ git merge iss53
```

```
Updating f483254..3a0874c
```

```
Fast forward
```

```
 README/      1-
```

```
 1 file changed, 0 insertions( + ), 1 deletions( - )
```

Узнать ветку, на которой мы находимся, можно используя команду `git status`.

1.2. Просмотр истории изменений с ветвлениями в удобном виде

Вы можете увидеть историю ревизий помощи команды `git log`. Команда `git log --oneline --decorate --graph --all` выдаст историю ваших коммитов и покажет, где находятся указатели ваших веток, и как ветвилась история проекта.

Пример 4. Просмотр изменений с ветвлениями в удобном виде:

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
/ * 87ab2 (testing) made a change
//
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

1.3. Разрешение конфликтов слияния

Иногда процесс не проходит гладко. Если вы изменили одну и ту же часть одного и того же файла по-разному в двух объединяемых ветках, Git не сможет самостоятельно объединить их. Если ваше исправление ошибки #53 потребовало изменить ту же часть файла, что и hotfix, вы получите примерно такое сообщение о конфликте слияния:

Пример 5. Конфликт при слиянии изменений:

```
$ 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 коммит слияния автоматически. Он остановил процесс до тех пор, пока вы не разрешите конфликт. Чтобы в любой момент после появления конфликта увидеть, какие файлы не объединены, вы можете запустить `git status`:

```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
```

Unmerged paths:

(use "`git add <file>...`" to mark resolution)

both modified: index.html

no changes added to commit (use "`git add`" and/or "`git commit -a`")

Всё содержимое, где есть неразрешенные конфликты слияния, перечисляется как неслитое. Git добавляет в конфликтующие файлы стандартные пометки разрешения конфликтов, чтобы вы могли вручную открыть их и разрешить конфликты.

В вашем файле появился раздел, выглядящий примерно так:

Пример 6. Файл с маркерами конфликта:

```
<<<<<<< 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
```

Способы разрешения конфликта в git такие же, как и в Subversion:

- Применить чужое изменение:

git merge -X theirs branch;

- Применить свои изменения:

git merge -s ours;

- Вручную объединить файлы (посредством ручного редактирования файла с метками конфликта).

Как только вы завершили слияние, зафиксируйте изменения в репозитории командой `git commit`.

2. Серверная работа с Git

2.1. Совместная работа

Самый простой метод совместной работы над проектом GitLab — это выдача другому пользователю прямого доступа на запись (операция `push`) в git-репозиторий. Вы можете добавить пользователя в проект в разделе “Участники” (“Members”) настроек проекта, указав уровень доступа. Получая уровень доступа “Разработчик” (“Developer”) или выше, пользователь может отсылать свои коммиты и ветки непосредственно в репозиторий.

Для добавления зайдите в настройки проекта `Settings-Members`. Выберите способ участия `Developer`. Дату окончания участия оставьте на своё усмотрение. Внешний вид окна настроек проекта представлен на рисунке 1. Внешний вид окна, возникающего при добавлении нового участника в существующий проект, показан на рисунке 2.

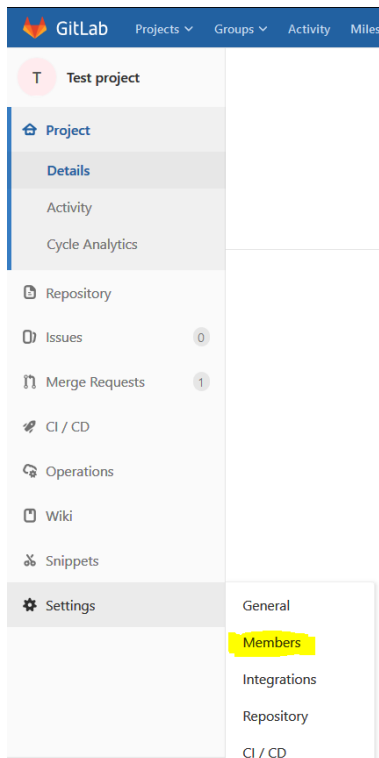


Рисунок 1 – Настройки проекта

Project members

You can invite a new member to **Test project** or invite another group.

Invite member	Invite group
Select members to invite	
<input type="text" value="Administrator x"/>	
Choose a role permission	
<input type="text" value="Developer"/>	
Read more about role permissions	
Access expiration date	
<input type="text" value="2018-12-26"/>	
<input type="button" value="Add to project"/> <input type="button" value="Import"/>	

Рисунок 2 – Окно добавления участника в проект

2.2. Создание репозитория

Рассмотрим пример создания нового проекта.

В окне создания проекта можно настроить его имя, URL, описание проекта, является ли он публичным и приватным. Также вам будет предложено проинициализировать проект, добавив в него файл

README. Внешний вид окна, возникающего при создании нового проекта, показан на рисунке 3.

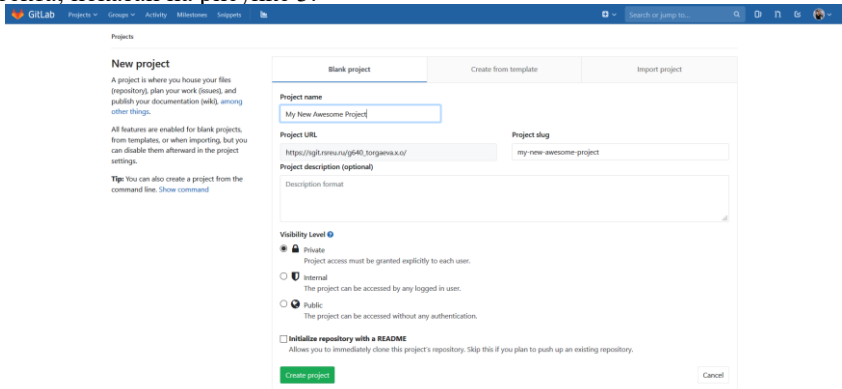


Рисунок 3 – Окно создания нового проекта

Добавление файлов в репозиторий через GitLab.

Для добавления файла зайдите в созданный проект (Project -> Details) и нажмите «+», Upload file (загрузить файл), и выберите нужный файл. Пример добавление нового файла показан на рисунке 4.

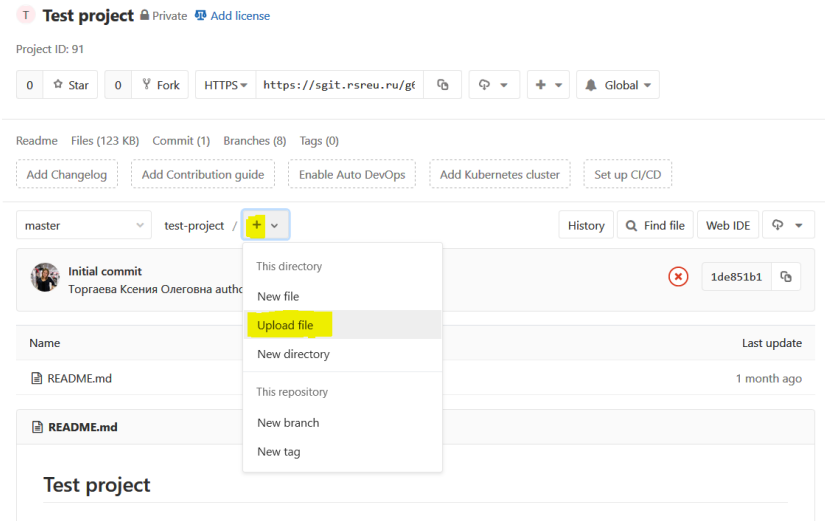


Рисунок 4 – Добавление файла через GitLab

2.3. Fork (форк) проекта

Можно воспринимать форк проекта как копию чужого репозитория у себя. Чтобы сделать форк в GitLab, нужно просто перейти на страницу с проектом, и нажать «Fork». Создание форк-а проекта



Рисунок 5 – Fork проекта

2.4. Клонирование проекта (создание рабочей копии)

Сразу после этого можно будет создать рабочую копию на своём компьютере и работать с проектом – для этого перейдите на страницу с проектом и нажмите “clone”, скопируйте url и создайте рабочую копию репозитория в выбранной вами папке с помощью команды **git clone <url>**.

2.5. Изменение последнего сообщения коммита

Бывают ситуации, когда вы неверно написали сообщение к коммиту. Вы можете установить сообщение к последнему коммиту непосредственно в командной строке:

Пример 7. Редактирование сообщение последнего коммита:

```
git commit --amend -m "New commit message"
```

Перед тем, как сделать это, убедитесь, что у вас нет каких-либо изменений в рабочей копии, иначе они также будут совершены. (Нестационарные изменения не будут зафиксированы.)

2.6. «Забирание» отдельного коммита

Команда **git cherry-pick** используется для того, чтобы взять изменения, внесённые каким-либо коммитом, и попытаться применить их заново в виде нового коммита наверху текущей ветки. Это может оказаться полезным чтобы забрать парочку коммитов из другой ветки без полного слияния с той веткой.

Пример 8. «Забирание» коммита с хеш-кодом «e43abfd3e...» в текущую ветку:

```
$ git cherry-pick e43abfd3e94888d76779ad79fb568ed180e5fcd
```

```
Finished one cherry-pick.
```

```
[master]: created a0a41a9: "More friendly message when locking the index fails."
```

```
3 files changed, 17 insertions(+), 3 deletions(-)
```

Практическая часть

Практическая часть:

Замечание: замените * на номер вашего варианта. Забирайте необходимые изменения у напарника и отправляйте их на удалённый сервер, когда необходимо.

1. Один из вас создаёт репозиторий, добавляет туда файла **fixme.cpp** (находится в этой же директории ЛР_4).

2. Участник, создавший репозиторий, добавляет второго в коллабораторы проекта через GitLab.

3. Первый и второй участник клонируют проект на компьютеры.

4. Создайте ветки по шаблону **feature/variant_*_ [имя]**.

5. Поправьте файл "fixme.cpp". Сделайте так, чтобы ваши изменения покрывали одни и те же участки кода.
6. Влейте изменения ветки напарника к себе в ветку. Отрадите в сообщении к коммиту, то, что это коммит с вливанием веток.
7. Разрешите возникшие конфликты.
8. Исследуйте, когда конфликт получается, а когда - нет.
9. Влейте изменения в ветку feature/develop_* (предварительно её создав).
10. Сделайте ещё два любых коммита в своих ветках.
11. Произведите редактирование последнего сообщения коммитов;
12. Влейте в свою ветку изменения напарника и затем отмените коммит со слиянием веток.

Содержание отчёта

По результатам выполнения работы оформляется отчет в соответствии с требованиями ГОСТ 7.32-2017 «Отчет о научно-исследовательской работе. Структура и правила оформления», включающий:

- титульный лист;
- цель работы;
- описание структуры хранилища во время выполнения (при выполнении операций, меняющих состояние хранилища);
- выполняемые команды с комментариями и результаты их выполнения;
- выводы.

Контрольные вопросы:

1. Опишите отличия локального репозитория от удалённого.
2. Назовите способы копирования проекта к себе в удалённый репозиторий.
3. Как отредактировать сообщение у коммита?
4. Что такое ветка?
5. Что представляет из себя ветка в Git?
6. Недостатки работы в одной ветке.
7. Как скопировать отдельные изменения между ветками?
8. Как скопировать изменения из одной ветки в другую?
9. Какие файлы создаются при конфликтной ситуации при слиянии?
10. Способы разрешения конфликтов в Git.

Лабораторная работа № 5. Работа с задачами через GitLab.

Цель работы

Получение навыков работы с системой отслеживания ошибок на примере GitLab Issue Tracking.

Внимание: для выполнения лабораторной вам будет необходимо установить IDE IntelliJ IDEA. Подробные инструкции приведены в документе «Настройка ПО для УРПО».

1. Теоретическая часть

1.1. Система отслеживания ошибок

Система отслеживания ошибок (англ. bug tracking system) — прикладная программа, разработанная с целью помочь разработчикам программного обеспечения (программистам, тестировщикам и прочим) учитывать и контролировать ошибки (баги), найденные в программах, пожелания пользователей, а также следить за процессом устранения этих ошибок и выполнением или невыполнением пожеланий.

В программировании *баг* (англ. bug — жук) — жаргонное слово, обычно обозначающее ошибку в программе или системе, которая выдаёт неожиданный или неправильный результат.

1.2. Состав информации о дефекте

Главный компонент системы отслеживания ошибок — база данных, содержащая сведения об обнаруженных дефектах. Эти сведения могут включать в себя:

- номер (идентификатор) дефекта;
- кто сообщил о дефекте;
- дата и время, когда был обнаружен дефект;
- версия продукта, в которой обнаружен дефект;
- серьёзность (критичность) дефекта и приоритет решения;
- описание шагов для выявления дефекта (воспроизведения неправильного поведения программы);
- кто ответственен за устранение дефекта;
- обсуждение возможных решений и их последствий;
- текущее состояние (статус) дефекта;
- версия продукта, в которой дефект исправлен.

Кроме того, развитые системы предоставляют возможность прикреплять файлы, помогающие описать проблему (например, дампы памяти или снимки экрана).

1.3. Жизненный цикл дефекта

Как правило, система отслеживания ошибок использует тот или иной вариант «жизненного цикла» ошибки, стадия которого определяется текущим состоянием, или статусом, в котором находится ошибка.

Типичный жизненный цикл дефекта:

1. Новый — дефект зарегистрирован тестировщиком.
2. Назначен — назначен ответственный за исправление дефекта.

3. Разрешён — дефект переходит обратно в сферу ответственности тестировщика.

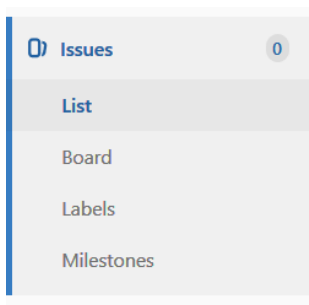
2. Работа с GitLab

2.1. Fork проекта

Можно воспринимать форк проекта как копию чужого репозитория у себя. Чтобы сделать форк в GitLab, нужно просто перейти на страницу с проектом, и нажать «Fork».

2.2. Заведение дефекта.

Для заведения дефекта во вкладке проекта выберите «Issues», затем «New Issue». Внешний вид требуемого элемента показан на рисунке 6.



The Issue Tracker is the place to add things that need to be improved or solved in a project

Issues can be bugs, tasks or ideas to be discussed. Also, issues are searchable and filterable.

New Issue

Рисунок 6 – Вкладка «Issues» проекта в GitLab

Создание новой задачи

Ниже рассмотрим пример заведения задачи/баги в трекере.

Во-первых, необходимо проставить название задачи, затем её описание.

Во-вторых, в лабораторной вам также нужно будет указать Assignee: человека, на которого назначена задача.

В-третьих, due date – время до следующей лабораторной. Остальные поля заполняются по желанию. Пример заведения задание в Issue Tracker'е сервиса GitLab показан на рисунке 7. Вид уже заведенной задачи продемонстрирован на рисунке 8.

New Issue

Title

Add [description templates](#) to help your contributors communicate effectively!

Description

Write Preview

B I

В методе **** ошибка, поправьте! Ничего не работает, сроки горят!

Markdown and [quick actions](#) are supported [Attach a file](#)

☐ This issue is confidential and should only be visible to team members with at least Reporter access.

Assignee Due date

Milestone

Labels

[Submit issue](#) [Cancel](#)

Рисунок 7 – Пример заведения ошибки в Issue Tracker'e GitLab

[Open](#) Opened 21 seconds ago by **Торгаева Ксения Олеговна**

[Close issue](#) [New issue](#)

Новая бага

В методе **** ошибка, поправьте! Ничего не работает, сроки горят!

0 0

[Create merge request](#)

Write Preview

B I

Write a comment or drag your files here...

Markdown and [quick actions](#) are supported [Attach a file](#)

[Comment](#) [Close issue](#)

Рисунок 8 – Вид заведённой задачи

2.3. Создание ветви для дефекта

Далее вам будет нужно создать ветку для решения этой задачи. Это можно сделать на странице с задачей, в окне «Create branch» (рисунок 9).

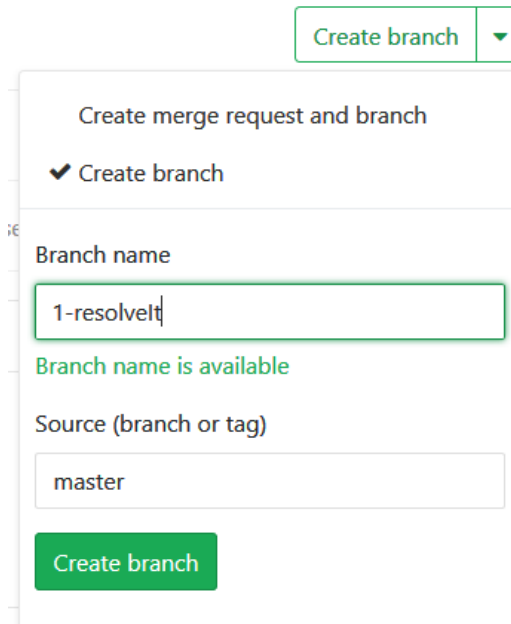


Рисунок 9 – Создание ветки, соотносящейся с задачей

2.4. Клонирование репозитория

Операции клонирования репозитория, перехода на ветку, коммита и отправки на удалённый сервер рассматривались ранее, поэтому здесь мы их рассмотрим кратко.

Для клонирования используется команда `git clone <URL>` (рисунок 10). URL можно найти на странице с проектом (Fork), который вы сделаете.

```
$ git clone https://sgit.rsreu.ru/g640_torgaeva.x.o/test-project.git
Cloning into 'test-project'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
```

Рисунок 10 – Клонирование проекта

1.4.5. Прочие необходимые операции

Далее вам нужно будет перейти на нужную ветку командой `git checkout <branch>`, где `branch` – ветка, которую вы создали. Затем исправить ошибку, описанную вашим напарником.

Далее нужно отправить изменения на удалённый сервер. Для этого добавьте изменённый файл к будущему коммиту (`git add <filename>`), и

сделайте коммит (`git commit -m «Осмысленное сообщение коммита»`) (рисунок 11). Далее отправьте изменение на удалённый сервер с помощью команды `git push` (рисунок 12).

```
$ git commit -m "Исправление баги"
[1-resolveIt 80867b2] Исправление баги
1 file changed, 1 insertion(+)
create mode 100644 file.txt
```

Рисунок 11 – Фиксирование изменений

```
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 322 bytes | 107.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: To create a merge request for 1-resolveIt, visit:
remote:   https://sgit.rsreu.ru/g640_torgaeva.x.o/test-project/merge_requests/new?merge_request%5Bsource_branch%5D=1-resolveIt
remote:
To https://sgit.rsreu.ru/g640_torgaeva.x.o/test-project.git
   1de851b..80867b2  1-resolveIt -> 1-resolveIt
```

Рисунок 12 – Отправка изменений на удалённый сервер

Практическая часть

Замечание: замените * на номер вашего варианта.

Ход работы:

1. Сделайте форк из репозитория <https://lexie62rus@bitbucket.org/lexie62rus/otslezhivanie-oshibok.git>.

2. Перейдите в нужный пакет (`util/variant_*`). Модули `Parser 1` и `Parser 2` для вас и напарника соответственно. Вы и ваш напарник, найдите ошибку в выбранном методе (одна ошибка в методе, см. таблицу 1 ниже). Ошибки можно вычислить по логике, или по несовпадению кода с комментарием к нему.

3. Заведите задачу в `GitLab`, и назначьте ее исполнение на напарника.

4. Создать ветку с любым именем на странице с заведённой задачей `issue tracker'a`.

5. Клонировать проект на локальный компьютер.

6. Исправьте ошибку в этой созданной ветке, проверьте и добавьте скриншоты и примеры.

7. Сделайте коммит. Залейте свои изменения на удалённый сервер.

8. Закройте задачу через `GitLab`.

Таблица 1 – Варианты заданий (названия методов с ошибками)

Вариант	Метод
	Calc (1.1 и 1.2)
	fixNegativeValue (2.1. и 2.2)
	deleteSpaces (3.1) isOperationSign (3.2)
	fixMultipleSignLack (4.1. и 4.2)
	openBracketsAndCalculateExpression (5.1. и 5.2)

Содержание отчёта

По результатам выполнения работы оформляется отчет в соответствии с требованиями ГОСТ 7.32-2017 «Отчет о научно-исследовательской работе. Структура и правила оформления», включающий:

- титульный лист;
- цель работы;
- описание хода выполнения работы с подробными комментариями и полученные результаты;
- выводы.

Контрольные вопросы:

1. Что такое система отслеживания ошибок?
2. Что такое баг?
3. Что такое crash report?
4. Опишите основные составляющие информации о дефекте.
5. Опишите жизненный цикл дефекта.
6. Назовите несколько известных вам систем отслеживания ошибок.
7. Опишите последовательность работы с дефектом с использованием GitLab.

СОДЕРЖАНИЕ

Лабораторная работа № 1. Subversion. Основные операции.....	1
Лабораторная работа № 2. Subversion. Ветвления.	8
Лабораторная работа № 3. Git. Основные операции.	13
Лабораторная работа № 4. Git. Серверная часть.	21
Лабораторная работа № 5. Работа с задачами через GitLab.....	28