РПОСУ. ЛР № 2. Система контроля версий Git

Цель работы

1. Знать понятия и компоненты систем контроля версий (СКВ), порядок и приемы работы с ними.

2. Уметь участвовать в командной разработке, используя конкретную СКВ — Git, а также популярный хостинг репозитариев — GitHub.

Выполнение работы

Выполнять необходимо все действия. Для удобства по тексту отмечены пункты, способ выполнения которых нужно придумать **самостоятельно.**

Нужно читать и осмысливать текст, который печатается в ответ на команды.

Отчет должен содержать все введенные команды, ответы системы, пояснения, что было сделано между пунктами (например, отредактирован файл), а также то, что требуется добавить в отчет по тексту задания.

Вход в терминал и создание структуры каталогов

Большая часть работы будет выполняться в терминале (командной строке). Для Windows вместе с Git поставляется программа Git Bash: эмулятор терминала Linux. Ее можно запустить из контекстного меню любого каталога пунктом *Git Bash Here* или из меню «Пуск».

Самостоятельно. Создайте на рабочем столе каталог `lab02` для данной ЛР и запустите в нем Git Bash.

Внимание. Даже если работа выполняется в компьютерном классе, создавайте каталог на рабочем столе, а не на сетевом диске, так как git не сможет нормально с ним работать.

В терминале откроется *приглашение (prompt)* примерно такого вида:

```
user@mpei-dc-win7 MINGW32 /c/Users/user/Desktop/lab02
$
```

Здесь важен рабочий каталог `/c/Users/user/Desktop/lab02` и символ `$` — начало ввода команд.

Просмотреть файлы в рабочем каталоге можно командой `ls`. В каталоге `lab02` пусто, поэтому `ls` ничего не выведет.

В ходе работы будем имитировать проект с двумя участниками: Алисой и Бобом. Компьютеры Алисы и Боба имитируют папки `lab02/alice` и `lab02/bob`:

```
mkdir alice
mkdir bob
```

Переход между каталогами делается командой `cd`. Перейдем «на компьютер Алисы» — в каталог `alice`:

```
cd alice
```

Самостоятельно. Создайте здесь каталог `project` и перейдите в него.

Перейти на уровень выше можно командой `cd ..` (две точки в конце, после `cd` пробел).

Самостоятельно. Перейдите из каталога проекта вверх, затем вернитесь в каталог `project`.

Все команды нужно заносить в отчет. Текст в Git Bash копируется при выделении, ничего нажимать не нужно. Скопированное можно вставить в любую другую программу *Ctrl+V,* как обычно.
Для вставки в сам Git Bash из буфера обмена нажмите правую кнопку мыши или *Ctrl+Insert.*

Инициализация репозитария и настройка Git

Инициализируем репозитарий в текущем каталоге (`project`):

```
git init
```

К приглашению командной строки добавилось `(master)`: имя текущий ветви Git. Ветвь `master` используется по умолчанию.

Git хранит свои данные в каталоге `.git` в корне репозитария — той папке, в которой сделано `git init`. Ее можно увидеть командой `ls -A`. Заходить в `.git` и что-либо делать там не нужно. Если удалить этот каталог, репозитарий будет безвозвратно утерян.

Git хранит три набора настроек:

- Системные — для всех пользователей компьютера, как правило, их не меняют.
- Пользовательские — для данного пользователя системы (`user` в примере). На практике чаще всего пользуются ими. Хранятся в профиле пользователя.

- Локальные — для отдельного репозитария, хранятся в нем же. Используются, если нужно работать с проектами от разного имени (примеры: личные и корпоративные проекты отдельно, Алиса и Боб в случае ЛР).

Локальные настройки имеют приоритет перед пользовательскими, а те — перед системными. Настройки Git сами не находятся под контролем версий, они специфичны для конкретного компьютера или конкретной копии репозитария.

Настроим репозитарий Алисы, чтобы коммиты были от ее имени:

```
git config user.name 'Alice (IvanovII)'
git config user.email 'alice@example.com'
```

Самостоятельно. Укажите данные пользователя, как в примере. Вместо `IvanovII` используйте свое имя и инициалы латиницей. Почта может быть любой, например, университетской, писем на нее не высылается.

Кавычки должны быть парными: или одинарными (`'`), или двойными (`"`). Если нарушить парность кавычек или нажать *Enter,* не закрыв кавычку, ввод команды продолжится на следующей строке. В этом случае можно прервать выполнение команды, нажав *Ctrl + C,* затем ввести команду правильно.

Создание коммитов

Запустите CodeBlocks и создайте проект в репозитарии Алисы. Убедитесь, что не создается ненужных подкаталогов:

Project title: `project`
Folder to create project in: `C:\Users\user\Desktop\lab02\alice`
Project filename: `project.cbp`
Resulting filename: `C:\Users\user\Desktop\lab02\alice\project\project.cbp`

Соберите проект.

На этом этапе должна быть следующая структура файлов и каталогов:

```
lab02
├── alice
│   └── project  <--------- текущий рабочий каталог
│       ├── .git <--------- создан командой "git init"
│       ├── bin  <--------- создан CodeBlocks при сборке
│       ├── obj  <--------- (то же самое)
│       ├── main.cpp    <-- код программы
│       └── project.cbp <-- файл проекта
└── bob
```

Занесение файлов под контроль версий

Вернувшись в Git Bash, просмотрим состояние рабочей копии:

```
git status
```

В отчете нужно пояснить, что означает каждая строка вывода этой команды.

Добавим файл `main.cpp` в отслеживаемые (в индекс):

```
git add main.cpp
```

На Windows может отобразиться такое сообщение:

```
warning: LF will be replaced by CRLF in main.cpp
The file will have its original line endings in your working directory
```

Оно безвредно. Смысл в том, что Git хранит файлы с немного измененном виде, чтобы обеспечивать удобную работу с репозитарием в любой операционной системе.

Самостоятельно. Еще раз просмотрите состояние рабочей копии и поясните в отчете изменения.

Выполним коммит с файлом `main.cpp` и коротким сообщением:

```
git commit -m 'code: заготовка программы'
```

Составление сообщений к коммитам

На практике важно, чтобы описания коммитов были информативными: в будущем по ним быстро читают историю проекта, ищут коммиты по ключевым словам. Заголовок (первая строка) должен быть коротким (желательно до 50 символов) и описывать суть изменений, потому что только он показывается в списке коммитов. Часто в заголовок включают тему (к какой части проекта относится коммит) или номер задачи в системе отслеживания ошибок:

- `code: заготовка программы` — изменен код (а не документация, например)
- `build: update CMake version` — коммит относится к сборке
- `обрабатывает пустой массив | fixes #1234` — исправляет ошибку № 1234
- `timer: учет високосных лет #4321` — доработка таймера по задаче № 4321

Временным незаконченным коммитам иногда приписывают `WIP:` (work in progress).

Из заголовка должно быть ясно, что в целом сделано и зачем (почему, для чего). Обычно нет смысла писать, какие именно файлы и функции были изменены, потому что это можно просмотреть в самом коммите. После заголовка через пустую строку может идти расширенное пояснение (тело), иногда очень длинное.

Как и для исходного кода, главный критерий — понятность и единообразие. Можно писать на русском или английском, в совершенной форме или в повелительном наклонении — но одинаково во всех коммитах.

Самостоятельно. Добавьте файл `project.cbp` в индекс и сделайте коммит с ним, тема — `build`. Сообщение после темы придумайте по смыслу изменений, например, для этого коммита подошло бы «добавлен файл проекта» или «add project file».

Создание коммитов с изменениями

Заменим тело функции `main()` на ввод двух чисел:

```
cout << "Enter A and B: ";
int a, b;
cin >> a >> b;
```

Самостоятельно. Просмотрите состояние репозитария (`git status`). В отчете поясните различия между случаем, когда добавлялся новый файл, и когда изменился существующий.

Чтобы закоммитить изменения, есть три способа, описанных ниже. Обратите внимание: Git «видит» состояние файлов на диске, поэтому после добавления изменений нужно сохранять файл в CodeBlocks. Желательно также собирать программу после изменений.

Способ 1. Сначала выбрать файлы, изменения которых должны войти в коммит, затем сделать коммит:

```
git add main.cpp
git commit -m "..."
```

Этот способ удобен, если изменения присутствуют не только в тех файлах, которые коммитятся. Например, если работа над кодом уже закончена, а документация еще не дописана и коммитить ее не нужно.

Самостоятельно. Добавьте в программу вывод суммы `a` и `b`.

Способ 2. Добавить в индекс все изменения, затем сделать коммит:

```
git add -u
git commit -m "..."
```

Способ удобен, если измнено много файлов. После `git add -u`, которая добавляет в индекс измененные файлы, можно командой `git add <файл>` добавить в индекс новые файлы.

Самостоятельно. Добавить в программу вывод разности `a` и `b`.

Внимание. Код доработок должен быть составлен в точности так:

```
cout << "A + B = " << a + b << '\n'
     << "A - B = " << a - b << '\n';
```

Дальнейшие дополнения тоже должны продолжать одну большую инструкцию вывода, а не быть отдельными. Это нужно для того, чтобы в последующих пунктах можно было наблюдать некоторые примечательные ситуации.

Способ 3. Добавить все изменения в индекс и сделать коммит в один шаг:

```
git commit -a -m "..."
```

Способ полностью эквивалентен предыдущему и удобен, если коммит меняет только существующие файлы.

Игнорирование файлов

Можно заметить, что в выводе команды `git status` все время присутствуют каталоги `bin/` и `obj/`. Они содержат бинарные файлы (целевой `*.exe` и промежуточные), которые являются производными от исходного кода, уже находящегося под контролем версий. Является грубой ошибкой заносить под контроль версий продукты сборки. То же самое относится к файлам, в которых некоторые среды сохраняют, например, состояние редактора: открытые файлы и расположение окон одного члена команды не нужны в общем хранилище.

Укажем Git игнорировать присутствие каталога `bin`. Для этого создадим в CodeBlocks новый файл *(File → New… → Empty)* и запишем в него строку:

```
/bin
```

Косая черта в начале означает путь от корня репозитария (каталога `project`), без нее игнорировался бы файл или каталог `bin` в любой подпапке. Сохраним файл в корне репозитарий под именем `.gitignore`, именно с точкой в начале.

Каждое правило игнорирования пишется на отдельной строке `.gitignore`.

Выполнив `git status`, можно видеть, что каталог `bin` не отображается.

Самостоятельно. Занесите каталог `obj` в список игнорируемых и убедитесь, что это удалось.

Файл `.gitignore` может и обычно должен находиться под контролем версий.

Самостоятельно. Создайте коммит с `.gitignore`, тема — `git`.

Просмотр истории

Работа с журналом репозитария

Журнал репозитария показывает команда `git log`. У нее много опций, например:

- `git log --stat` показывает файлы, измененные в коммитах;
- `git log --oneline --decorate` показывает коммиты компактно;
- `git log --oneline --decorate --all --graph` делает то же для всех веток.

Среди прочего, команда показывает для каждого коммита его хэш, например, `d2e8af7ff9c4684d0deb60d3305474bcaf69ce5c`. Некоторые версии команды показывают хэш сокращенно — краткий вариант тоже будет восприниматься командами Git, которые принимают хэш.

Если лог изменений длинный, `git log` показывает текст с прокруткой. Чтобы выйти из этого режима, нажмите `q`.

Попробуйте каждую из приведенных команд. **В отчете** подробно опишите, что показывается `git log --stat` для последнего коммита.

Коммиты можно фильтровать по разным признакам:

- `git log -- main.cpp` показывает затрагивающие `main.cpp`;
- `git log --grep "code:"` показывает коммиты с `code:` в сообщении.

Самостоятельно. Найдите сначала коммиты по теме `build`, затем коммиты, затрагивающие `project.cbp`.

Просмотр коммитов

Содержимое отдельных коммитов просматривается командой `git show <refspec>`, где `<refspec>` может быть хэшем коммита, именем ветви или выражением, которое задает, на сколько от них отступить в истории.

Просмотрим последний коммит тремя эквивалентными способами:

1. `git show HEAD` (текущий)
2. `git show master` (по имени ветви)
3. `git show d2e8af` (по хэшу нужного коммита)

Для просмотра предыдущего коммита можно либо записать его хэш, либо указать, что от последнего нужно отступить на один коммит: `HEAD~1`.

Самостоятельно. Просмотрите предпоследний коммит **(в отчете** зафиксируйте результат единожды) тремя способами.

Просмотр изменений

Внесем изменения в `main.cpp`: добавим печать произведения чисел, но не станем пока делать коммит.

Просмотрим изменения в рабочей копии:

```
git diff
```

В отчете необходимо пояснить все компоненты отображаемого патча.

Первый аргумент команды `git diff` включает показ изменений от указанного коммита до последнего, включая изменения в рабочей копии:

```
git diff HEAD~2
```

С двумя аргументами команда показывает разницу между указанными коммитами, например, так можно исключить изменения в рабочей копии из вывода предыдущей команды:

```
git diff HEAD~2 HEAD
```

Самостоятельно. Просмотрите изменения между самым первым коммитом и коммитом, добавляющим вывод разности.

Использование GUI

Просмотр истории — одна из операций в СКВ, которую иногда удобнее выполнять из графической среды. Вместе с Git поставляется графическая оболочка gitk (Git GUI), которую можно вызвать пунктом *Git GUI Here* в контекстном меню папки проекта. Эта оболочка очень примитивная, на практике пользуются более мощными или встроенными в среду разработки.

Просмотр истории в gitk делается из меню *Repository* → *Visualize All Branch History.* Можно выбирать коммит для просмотра из списка; смотреть как изменения *(Diff),* так и версии файлов *(Old version, New version);* искать коммиты.

В отчет ничего заносить не нужно.

Откат изменений

Самостоятельно. Закоммитьте изменения в рабочей копии (вывод произведения).

Предположим, необходимо отменить (откатить) этот коммит, то есть вернуться к предыдущему. Для этого воспользуемся командной `git reset`:

```
git reset --hard HEAD~1
```

Здесь `HEAD~1` указывает на коммит, к которому нужно откатить состояние рабочей копии, а ключ `--hard` означает, что нужно привести рабочую копию точно к состоянию выбранного коммита.

CodeBlocks (и другие среды) могут при этом показать предупреждение, что файл на диске был изменен, и предложить загрузить его заново. Следует согласиться.

Добавим над функцией `main()` комментарий:

```
// you may type whatever you want
```

Уберем изменения в `main.cpp` другим способом — откатив этот файл к состоянию в последнем коммите (`HEAD`):

```
git checkout HEAD -- main.cpp
```

Второй способ необходим, чтобы откатывать отдельные файлы. Аргумент `HEAD` необязателен, но вместо него можно указать не последний, а любой другой коммит. Это полезно, если нужно восстановить состояние одного файла таким, какое оно было в известный момент.

Обмен кодом через удаленное хранилище

Регистрация на GitHub

Зарегистрируйтесь на GitHub под именем вида `KozlyukDA` (своя фамилия и инициалы).
При регистрации нужно указывать действующую почту — на нее придет письмо для подтверждения регистрации.

Настройка SSH

Отправлять изменения в удаленный репозитарий обычно могут не все. Так, доступ на запись к репозитариям на GitHub по умолчанию есть только у создателя (пользователя, зарегистрированного на предыдущем шаге). Загрузка комитов (доступ на чтение) из публичных репозитариев разрешена всем.

GitHub должен выяснить, что клиент, представившийся определенным пользователем, действительно им является (провести *аутентификацию*). Клиент git взаимодействует с сервером GitHub по протоколу SSH (secure shell), который использует для аутентификации пары ключей: открытый (public, публичный) и закрытый (private, приватный) ключ. Конкретный открытый ключ связан с конкретным закрытым. Сначала открытый ключ загружается на сервер GitHub через web-интерфейс. Затем любой клиент, который обладает соответствующим закрытым ключом, может доказать это серверу. У одного пользователя может быть несколько пар ключей, например, для рабочего и домашнего компьютера, тогда на GitHub загружаются два открытых ключа.

Внимание. Закрытый ключ является таким же секретом, как пароль пользователя. Любой, кто получит закрытый ключ, сможет вносить на сервер изменения от имени вашего пользователя. Закрытый ключ нельзя давать никому, открытый ключ можно давать свободно.

Создать пару ключей:

```
ssh-keygen
```

По умолчанию закрытый ключ записывается в файл `/home/user/.ssh/id_rsa`, можно оставить это значение по умолчанию (нажать *Enter*). Далее нужно ввести пароль, которым будет защищен ключ, и повторить его.

Пример вывода команды:

```
Generating public/private rsa key pair. Enter file in which to save the key
(/home/user/.ssh/id_rsa): (Enter)
Enter passphrase (empty for no passphrase): (ввод не отображается)
Enter same passphrase again: (ввод не отображается)
Your identification has been saved in /home/user/.ssh/id_rsa Your public key has been saved in
/home/user/.ssh/id_rsa.pub
(Далее следуют уникальные для каждого ключа строки.)
```

Вводить пароль каждый раз, когда используется ключ, неудобно. Используют программу-агент, которая работает в фоне и предоставляет ключи другим программам, в том числе git. Пароль требуется тогда вводить один раз — при загрузке ключа в агент.

Запустить агент:

```
eval `ssh-agent -s`
```

Загрузить ключ (потребуется ввести пароль):

```
ssh-add
```

По умолчанию `ssh-add` загружает `~/.ssh/id_rsa`, для загрузки других ключей, если это нужно, можно передавать ей путь к файлу ключа явно.

Отобразить открытый ключ можно командой:

```
cat ~/.ssh/id_rsa.pub
```

Самостоятельно. Скопировать открытый ключ (текст) и добавить в список открытых ключей своей учетной записи GitHub. Это делается в настройках (меню пользователя в правом верхнем углу, пункт *Settings*), раздел *SSH and GPG keys,* кнопка *New SSH key.*

Если работа выполняется в компьютерном классе, закрытый ключ будет утерян после выхода из учетной записи (или выключении компьютера). Проще всего дома или на следующем занятии создать новый ключ и добавить его на GitHub. На практике ключи не уничтожают (кроме случаев, когда их украли), а переносят как файлы, например, при переустановке системы. Из проводника Windows файл закрытого ключа виден как `C:\Users\User\.ssh\id_rsa`, если понадобится его скопировать.

Отправка проекта на GitHub

Создайте репозитарий под названием `cs-lab02`. Вопреки рекомендациям по ссылке, не нужно добавлять в репозитарий файл `README.md` или лицензию.

После создания пустого репозитария будет показана страница с инструкциями, как настроить связь с хранилищем на GitHub:

1. В разделе *Quick setup* нужно выбрать вариант SSH.
2. В разделе *...or push an existing repository from the command line* даны команды, которые необходимо выполнить.

При взаимодействии с удаленным хранилищем будет запрошено имя пользователя и пароль от GitHub.

Обновите страницу и убедитесь, что проект успешно загружен на GitHub. Любой файл можно просмотреть в бразуере. По ссылке *Commits* можно просматривать коммиты.

Получение проекта с GitHub

Предположим, к разработке проекта присоединяется Боб. Откройте новый терминал Git Bash в каталоге `bob`. Клонируйте проект:

```
git clone <адрес> <каталог>
```

На место `<адреса>` нужно подставить адрес, который использовался в команде `git remote add` (его можно всегда отобразить командой `git remote -v`). Каталог — название папки для проекта: используйте `project`, если не указывать, это было бы название репозитария (`cs-lab02`). Угловых скобок в команде быть не должно!

Перейдите в каталог проекта «на машине Боба» (здесь и далее это означает работу во втором терминале и над файлами в `bob/project`):

```
cd project
```

Самостоятельно. «На машине Боба» настройте Git (`git config`) аналогично тому, как это делалось для Алисы в начале лабораторной работы.

Совместная работа над проектом без конфликтов правок

«На машине Боба» добавьте в программу печать произведения чисел и сделайте коммит. Просмотрите последний коммит и убедитесь, что он сделан от имени Боба.

Отправьте коммит на GitHub (используйте те же учетные данные, что и ранее):

```
git push
```

Обновите страницу GitHub и убедитесь, что коммит попал в удаленный репозитарий. Обратите внимание, что авторство коммитов записано в самих коммитах, оно не зависит от пользователя системы или GitHub.

«На машине Алисы» (то есть в первом терминале, в каталоге `alice/project`) выполните загрузку изменений:

```
git fetch
```

Убедитесь, что в рабочей копии изменений еще не произошло.

Просмотрите историю всех веток:

```
git log --oneline --decorate --all --graph
```

Как можно видеть, ветка `master` отстает на один коммит от ветки `origin/master` (версии ветки `master` из удаленного репозитария под названием `origin`, то есть на GitHub).

Продвиньте ветку `master` к скачанной версии:

```
git pull --ff-only
```

Убедитесь, что рабочая копия проекта «у Алисы» соответствует версии «у Боба».

Команда `git pull` автоматически делает `git fetch`, поэтому можно было бы применять только ее, но важно понимать, что получение изменений в Git двухфазное: загрузка новой части истории и синхронизация положения веток.

Самостоятельно. «От имени Алисы» добавьте в программу печать деления, сделайте коммит, отправьте его на GitHub и получите новую версию «на машине Боба». Иначе говоря, повторите шаги выше, поменяв местами роли Алисы и Боба.

Разрешение конфликтов правок при совместной работе

Предположим, Алиса решает добавить в программу печать максимума из чисел, а Боб — минимума.

Внимание. Код вывода в программе перед выполнением дальнейшего должен иметь следующий вид:

```
cout << "A + B = " << a + b << '\n'
     << "A - B = " << a - b << '\n'
     << "A * B = " << a * b << '\n'
     << "A / B = " << a / b << '\n';
```

В противном случае код нужно привести в соответствие отдельным коммитом и синхронизировать состояние «у Алисы» и «у Боба».

«На машине Алисы» дополните программу печатью максимума, сделайте коммит и отправьте его на GitHub.

«На машине Боба» дополните программу печатью минимума, сделайте коммит и попытайтесь отправить его на GitHub. Как можно видеть, удаленный репозитарий не принимает изменений: коммит Боба основан не на последнем существующем коммите.

«От лица Боба» загрузите коммиты из удаленного хранилища и отобразите историю всех веток — результат нужно представить **в отчете.**

Можно видеть, что ветка `master` раздвоилась. Бобу нужно переместить свой коммит поверх коммита Алисы, то есть поверх `origin/master`:

```
git rebase origin/master
```

Однако эта команда завершается с ошибкой, сообщающей о конфликте в `main.cpp`. Просмотрите состояние хранилища и поясните **в отчете.**

«На машине Боба» в CodeBlocks место конфликта будет отмечено прямо в коде. Необходимо **самостоятельно:**

1. Удалить метки конфликта: `<<<< ...`, `... >>>>` и `=====`.
2. Отредактировать код так, чтобы он включал и правки Алисы, и правки Боба.
3. Убедиться, что программа компилируется и работает.

После того, как конфликт разрешен, нужно добавить файл в индекс и продолжить прерванную операцию `rebase`:

```
git add main.cpp
git rebase --continue
```

Убедитесь, что история хранилища теперь имеет желаемый вид (зафиксировав это **в отчете)** и отправьте изменения на GitHub.

Использование веток

Предположим, пока Боб синхронизировал изменения, Алиса решила изменить тип чисел с целых на действительные. Предполагая, что это займет время, Алиса ведет работу в отдельной ветке. На момент начала работы репозитарий Алисы *не* синхронизирован с GitHub, то есть последний коммит добавляет печать максимума. Все действия ведутся «на машине Алисы».

Создайте ветку `double`:

```
git branch double
```

Переключитесь на нее:

```
git checkout double
```

Примечание. Создание ветки и переключение на нее можно делать одной командой: `git checkout -b double`. Этой команде можно передать аргумент-ссылку на коммит, где создать ветку.

Можно заметить, что текущая ветка в приглашении терминала изменилась.

Замените тип переменных `a` и `b` на `double` и сделайте коммит.

Переключитесь на ветку `master`:

```
git checkout master
```

Самостоятельно. Синхронизируйте ветку `master` «на машине Алисы» с GitHub. Просмотрите историю всех веток и занесите результат **в отчет.**

Слейте ветку `double` в `master`:

```
git merge double
```

В результате слияния образуется специальный новый коммит (merge commit), к которому Git предлагает написать сообщение в редакторе. Строки, начинающиеся с октоторпа («решетки», `#`), в сообщение не войдут.

Отправьте изменения на GitHub.

Просмотрите и занесите **в отчет** историю всех веток репозитария.

Редактор Vim

Vim — продвинутый текстовый редактор. Он не является частью Git, но популярен в системах семейства *nix, поэтому предлагается по умолчанию. Не обязательно его использовать, но нужно знать минимум для обращения с ним.

После запуска Vim находится в так называемом *нормальном режиме.* Чтобы начать вводить текст, нужно перейти *в режим вставки,* нажав `i` (одну клавишу). В режиме вставки можно набирать текст обычным образом. Вернуться в нормальный режим можно нажатием *Escape.* Находясь в нормальном режиме, можно сохранить сообщение и выйти из Vim нажатием `ZZ` (две заглавные Z, то есть *Shift+Z* два раза).

Если вместо *Shift+Z* нажать *Ctrl+Z,* Vim будет приостановлен, а коммит останется незавершенным. В этом случае нужно вернуться в Vim командой `fg` в терминале.

Чтобы писать длинные сообщения, но не использовать Vim, можно указать другой редактор (например, примитивный `nano`):

```
EDITOR=nano git merge double
```

Формат защиты

Защита состоит из ответов на теоретические вопросы по лекции и выполнения задания, рассчитанного на 10 минут.

Пример задания:

1. Создать новый репозитарий.
2. Закоммитить файл `task.txt` с цифрами от 0 до 9 на отдельных строках.
3. Удалить строки с цифрами от 5 до 8, закоммитить.
4. Просмотреть предпоследний коммит.
5. Создать ветку `task` от предыдущего коммита.
6. Переключиться на ветку `task`.
7. Добавить в начало файла строки с буквами `a`, `b`, `c`, закоммитить.
8. Переключиться на ветку `master`.
9. Добавить в начало файла строки с буквами `d`, `e`, `f`, закоммитить.
10. Слить ветку `task` в `master`, разрешив конфликт так, чтобы буквы шли по порядку.

Козлюк Д. А., Мохов А. С. для кафедры Управления и интеллектуальных технологий НИУ «МЭИ», 2022 г.