

# **Software version control using Git**

## CONTENT

1 GETTING TO KNOW THE GIT DISTRIBUTED VERSION CONTROL SYSTEM .....	4
1.1 Preparation for work.....	4
1.2 Basics of Git .....	4
1.2.1 Casts of the file system.....	4
1.2.2 Local operations .....	6
1.2.3 Data integrity control .....	6
1.2.4 Data is just being added .....	7
1.2.5 File states .....	7
1.3 Performance of work.....	9
1.3.1 Initial Setup .....	9
1.3.2 Creating a repository .....	10
1.3.3 Recording changes in the repository .....	11
1.3.4 View commit history.....	21
1.3.5 Cancellation of changes .....	22
1.4 Requirements for the report.....	<b>Ошибка! Закладка не определена.</b>
1.5 Questions for self-testing .....	28
2 DOCUMENTING THE REQUIREMENTS AND DESIGNING THE SYSTEM ARCHITECTURE USING THE UML LANGUAGE. WORKING WITH BRANCHES IN THE GIT SYSTEM	<b>Ошибка! Закладка не определена.</b>
2.1 Preparation for work.....	<b>Ошибка! Закладка не определена.</b>
2.2 Documentation of requirements .....	<b>Ошибка! Закладка не определена.</b>

2.2.1 User Stories .....	<b>Ошибка! Закладка не определена.</b>
2.2.2 Usage scenarios .....	<b>Ошибка! Закладка не определена.</b>
2.2.3 Detailed requirements .....	<b>Ошибка! Закладка не определена.</b>
2.3 System architecture design.....	<b>Ошибка! Закладка не определена.</b>
2.4 Working with branches in the Git system .....	29
2.4.1 Branching and merging .....	29
2.4.2 Merger conflicts .....	35
2.5 Report requirements .....	<b>Ошибка! Закладка не определена.</b>
2.6 Questions for self-testing .....	38

# **1 GETTING TO KNOW THE GIT DISTRIBUTED VERSION CONTROL SYSTEM**

## **1.1 Preparation for work**

1. Create a directory on the disk (for example, D:\GIT\_PRACTICE) and place in it a subdirectory with files created in previous works, etc. (for example, D:\GIT\_PRACTICE\analysis).

2. Install Git, for which you need to download the installer exe file from the project page (<https://gitforwindows.org/>) and run it. After installation, both the console version and the standard graphical version will be available for use. It is recommended to use Git only from the command shell that is part of the installed system, because only in this way will all the commands used, including those in this work, be available.

To install Git in other operating systems, you must refer to the instructions:

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

## **1.2 Basics of Git**

### **1.2.1 Casts of the file system**

The main difference between Git and any other version control systems (for example, Subversion and similar) is the way in which data is organized in Git. Most other version control systems store data in the form of a list of changes (patches) for files. Such systems (CVS, Subversion, Perforce, Bazaar, and others) present stored data as a set of files and changes made to each of these files over time (Figure 1.1).

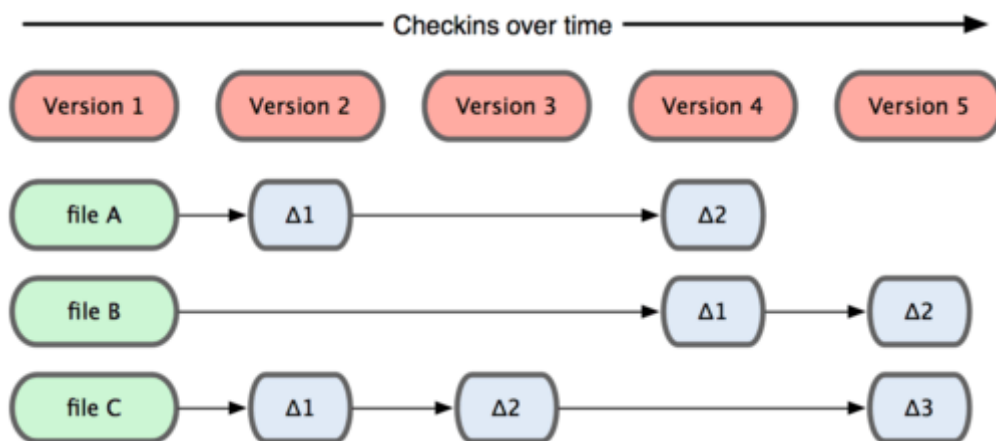


Figure 1.1

Instead of storing data in the form shown in Figure 1.1, Git represents stored data as a set of snapshots of a small file system. Every time a user commits a current version of a project, the Git version control system saves a snapshot of what all the project files look like at that moment. For efficiency, if the file has not been modified, Git does not save the file again, but creates a link to the previously saved file. This approach is depicted in Figure 1.2.

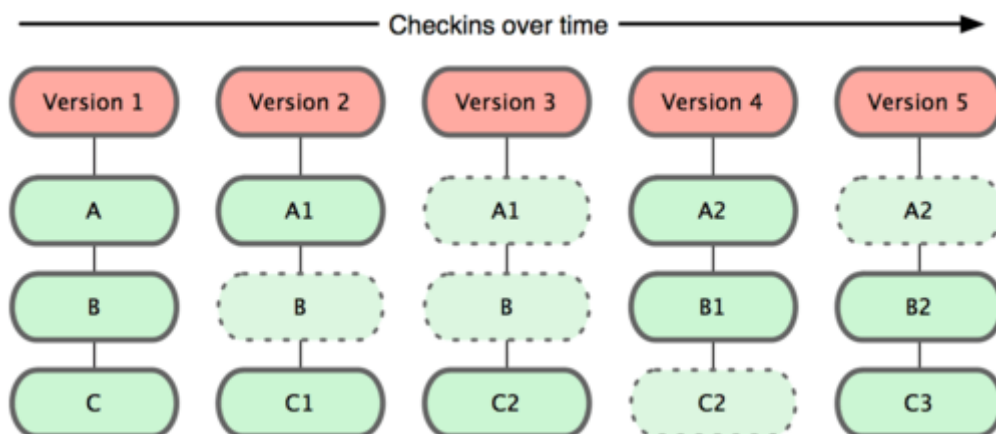


Figure 1.2

This feature distinguishes Git from almost all other version control systems. As a result, the creation of Git required a revision of almost all aspects of version control that other systems adopted from their predecessors. So Git is

more like a small file system with powerful tools running on top of it than a typical version control system.

### **1.2.2 Local operations**

For most operations in the Git system, only local files and resources are needed, that is, usually information from other computers on the network is not needed. Since the entire project history is stored locally on a disk dedicated to the computer user, most operations are performed almost instantly.

To demonstrate project history, Git does not download it from the server, but simply reads it directly from the local repository of the specific user who requested the project history to be demonstrated. If you need to see the changes between the current version of a file and a version made a month ago, Git can calculate the difference locally, instead of asking the version control server for the difference or downloading an old version of the file and only then doing the local comparison.

Local execution of operations means that only a small fraction of operations cannot be performed without network or VPN access. If the user wants to work without having access to the network, for example, while on a plane or train, he can continue to make commits (record changes to the project) and then send them to the server as soon as the network becomes available. Likewise, if the VPN client is down, you can still continue.

In many other version control systems, full-fledged local work is impossible or extremely inconvenient. For example, using Perforce, almost nothing can be done without a connection to the server. When working with Subversion and CVS, the user can edit files, but it is not possible to save the changes to the local database, because it is disconnected from the central repository.

### **1.2.3 Data integrity control**

Before any file is saved, Git calculates its checksum, which is used as the file's index. Therefore, it is impossible to change the contents of a file or

directory without the changes being detected by the Git system. This functionality is an important part of Git. If information is lost or corrupted in transit, Git will always detect it.

The mechanism used in Git to calculate checksums is called SHA-1 hashing. This is a string of 40 hexadecimal characters (0-9 and af) calculated based on the contents of the file or directory structure. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

When working with Git, such hashes are found everywhere, as they are very widely used in the Git system. In fact, Git stores everything in its database not by file names, but by hashes of their contents.

#### **1.2.4 Data is just being added**

Practically all actions performed by the user in Git only add data to the database. It is very difficult to force the system to delete data or do something that cannot be undone. It is possible, as with any version control system, to lose data that has not yet been committed, but once it is committed, it is very difficult to lose it, especially if changes are regularly pushed to a central repository. Therefore, when using the Git system, you can experiment without fear of seriously damaging something in the project.

#### **1.2.5 File states**

The most important thing to know about Git is that files can be in one of three states on the system:

- 1) "fixed" - the file is already saved in the local repository;
- 2) "changed" - a file that has been changed, but not yet fixed;
- 3) "prepared" - modified file marked for inclusion in the next commit.

Thus, there are three areas in projects using Git (Figure 1.3):

1) Git directory (Git directory) – the place where Git stores metadata and the database of objects of the user project; this is the most important part of Git and is copied when the repository is cloned from the server;

2) working directory – a copy of a certain version of the project extracted from the database; these files are pulled from the database in the Git directory and placed on disk so that they can be viewed and edited;

3) the area of prepared files (staging area) is a file usually stored in the Git directory, which contains information about what should be included in the next commit; sometimes called an index.

A standard workflow using the Git version control system looks something like this (Figure 1.3):

1. The user makes changes to the files in his working directory.
2. The user prepares files by adding their casts to the prepared files area.
3. The user makes a commit that takes the prepared files from the area of prepared files (index) and places them in the Git directory for permanent storage.



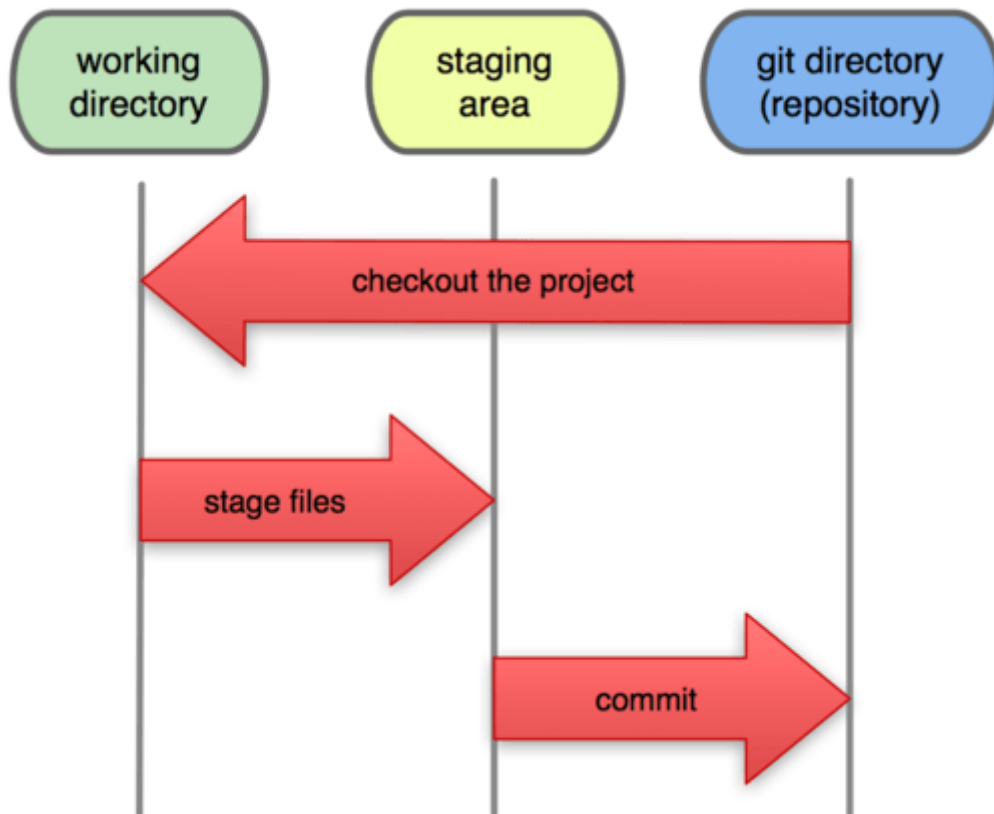


Figure 1.3

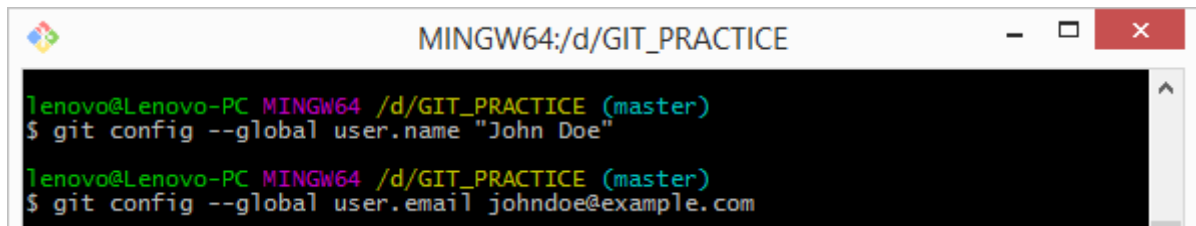
If the working version of a file matches the version in the Git directory, the file is considered committed. If the file is modified but added to the prepared data area, it is prepared. If the file changed after being downloaded from the database, but was not prepared, then it is considered changed.

## 1.3 Performance of work

### 1.3.1 Initial Setup

The Git system includes the `git config` utility, which allows you to view and set parameters that control all aspects of Git's operation and appearance.

The first thing to do after installing Git is to provide a name and email address. This is necessary because every commit contains this information, and it is included in the commits passed by the user and cannot be changed further. Start Git Bash and type (Figure 1.4):

A screenshot of a Windows terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The first command entered is '\$ git config --global user.name "John Doe"'. The second command entered is '\$ git config --global user.email johndoe@example.com'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git config --global user.name "John Doe"

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git config --global user.email johndoe@example.com
```

Figure 1.4

If the `--global` option is specified, then it is enough to make these settings only once. If you need to specify a different name or email for some individual projects, you can run the same commands without the `--global` option in the directory with the desired project.

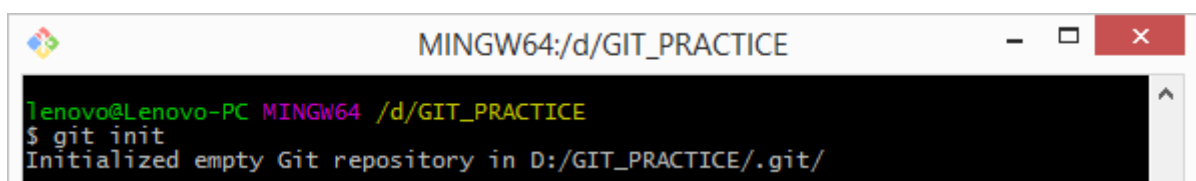
More information about Git setup can be found at:

<https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>

### 1.3.2 Creating a repository

There are two main approaches to creating a Git repository. The first approach is to import an already existing project or directory into Git. The second approach is to clone an existing repository from the server. As can be seen from point 1.1, this work will consider the first approach to creating a repository. Cloning an existing repository from a server will be covered later.

Go to the project directory (D: \ GIT\_PRACTICE) and in the command line (PCM, Git Bash Here item) enter (Figure 1.5):

A screenshot of a Windows terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE'. The command entered is '\$ git init'. The output is 'Initialized empty Git repository in D:/GIT\_PRACTICE/.git/'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE
$ git init
Initialized empty Git repository in D:/GIT_PRACTICE/.git/
```

Figure 1.5

The `git init` command creates a new subdirectory named `.git` in the current directory that contains all the necessary Git repository files. At this stage, the project is still not under version control.

### **1.3.3 Recording changes in the repository**

Since there is now a Git repository and a working copy of files for the project (`D:\GIT_PRACTICE\analysis`), it is necessary to make some changes and capture snapshots (or snapshots) of the state of these changes in the repository every time the project reaches a state that would wanted to save.

It must be remembered that each file in the working directory can be in one of two states:

1) "tracked" - a file that was in the last snapshot of the project status (which is under version control); it can be unchanged, changed or prepared for commits;

2) "not tracked" - any file in the working directory that was not included in the last state snapshot and is not prepared for commits.

When the repository is cloned, all files are tracked and they are unchanged because they were just cloned (checked out) but not edited.

As soon as files are edited, Git will treat them as modified. Changes must be indexed (prepared for commits) and then committed, after which the cycle repeats. This life cycle is depicted in Figure 1.6.

In our case, the project was imported, not cloned from another repository, so all files are untracked because they were not included in the last snapshot (no snapshot has been taken yet) and have not yet been prepared for commits.

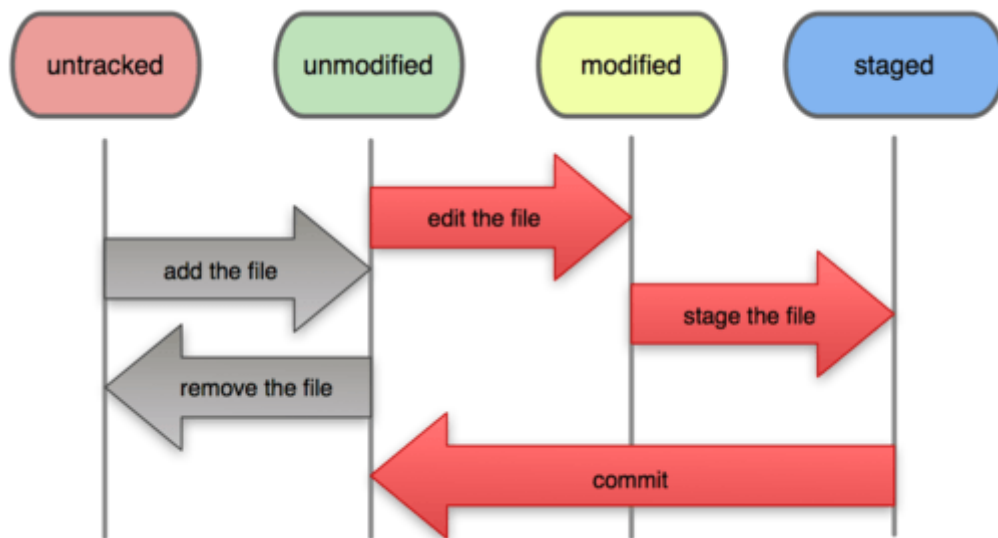


Figure 1.6

To determine which files are in which state, the `git status` command is used (Figure 1.7):

```
MINGW64:/d/GIT_PRACTICE

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master

Initial commit

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

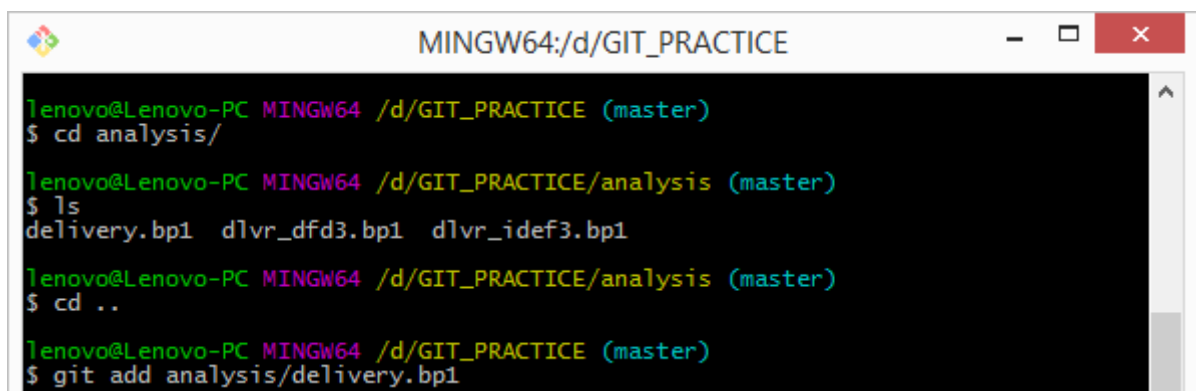
analysis/
```

Figure 1.7

You can understand that the `analysis` directory is not tracked by the fact that it is located in the "Untracked files" section in the status command output. In addition, the command informs the user which branch he is currently on. For now, it's always the master branch - it's the default branch. Features of working with branches will be considered later.

Git will not add untracked files to commits until the user explicitly tells them to do so. This prevents you from accidentally adding binaries or anything else that wasn't intended to be added to the repository.

To track a new file, use the git add command. To add one of the files of the analysis directory under version control, you must do the following (Figure 1.8):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing a series of commands and their outputs. The user navigates to the 'analysis' directory, lists files, and then adds 'delivery.bp1' to the repository.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ cd analysis/

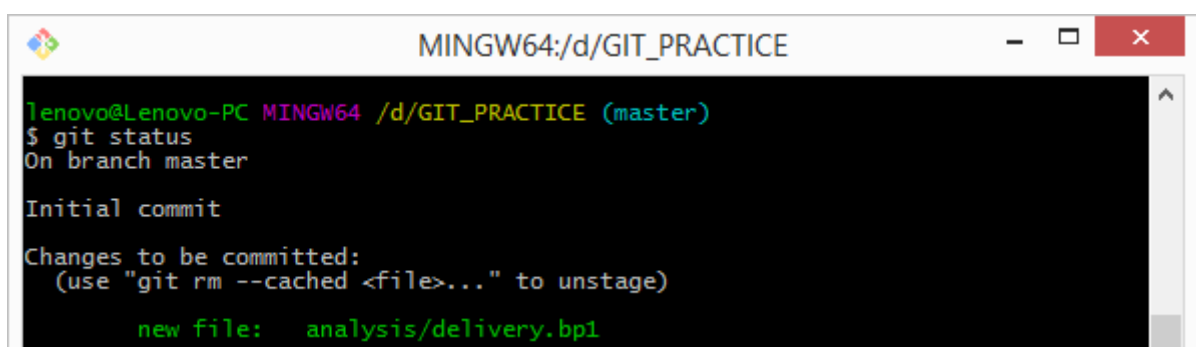
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE/analysis (master)
$ ls
delivery.bp1  dlvr_dfd3.bp1  dlvr_idef3.bp1

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE/analysis (master)
$ cd ..

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git add analysis/delivery.bp1
```

Figure 1.8

If you execute the status command again, you will see that the delivery.bp1 file is now tracked and indexed (Figure 1.9):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the output of the 'git status' command. It indicates an initial commit and shows that 'analysis/delivery.bp1' is a new file to be committed.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   analysis/delivery.bp1
```

Figure 1.9

It can be seen that the file is indexed because it is in the "Changes to be committed" section. If the commit is executed at this point, the version of the file that existed at the time of the git add command will be added to the snapshot

history. The git add command takes as a parameter the path to a file or directory, if it is a directory, the command recursively adds (indexes) all files in the given directory.

Add the remaining untracked files from the analysis directory using the git add command, check the index status using the git status command (Figure 1.10):

A screenshot of a terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The terminal shows the following commands and output:

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git add analysis/

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   analysis/delivery.bp1
        new file:   analysis/dlvr_dfd3.bp1
        new file:   analysis/dlvr_idef3.bp1
```

Figure 1.10

Most often, the project contains a group of files that not only do not need to be automatically added to the repository, but also need to be seen in the untracked list. Such files usually include automatically generated files (various logs, program compilation results, etc.). In this case, you can create a .gitignore file listing the templates of the corresponding such files. Detailed information on the formation of .gitignore can be obtained from the link:

<https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>

Create a hello.c file in the working directory, which, for some reason, should not be tracked (Figure 1.11):

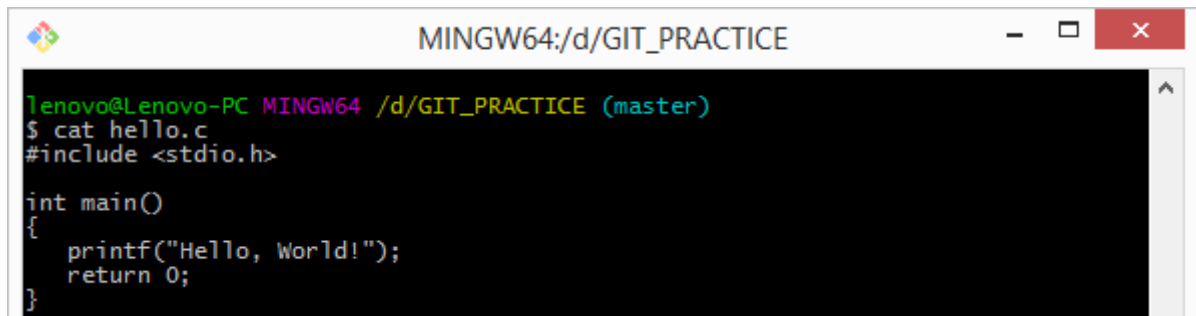
A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' with standard window controls. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The command '\$ cat hello.c' has been executed, displaying the content of the file: '#include <stdio.h>', 'int main()', '{', 'printf("Hello, World!");', 'return 0;', and '}'.

Figure 1.11

Create a file with the name `.gitignore` and the following content, to exit the file content input mode, press `Ctrl + D` (Figure 1.12):

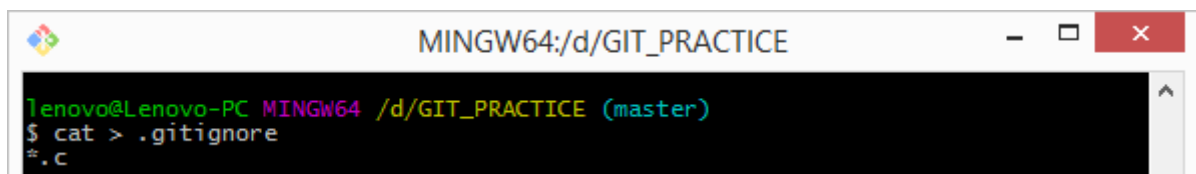
A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' with standard window controls. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The command '\$ cat > .gitignore' has been entered, and the first line of the file, '\*.c', is visible.

Figure 1.12

A single line in the generated `.gitignore` file tells Git to ignore any files ending in `.c`. The list can also include directories to be ignored by ending the pattern with a slash (`/`) to indicate the directory. Empty lines and lines starting with `#` (comments) are ignored.

Check the correctness of the creation of the `.gitignore` file using the `git status` command (Figure 1.13):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the output of the 'git status' command. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The output indicates an initial commit on the master branch. It lists three new files to be committed: 'analysis/delivery.bp1', 'analysis/dlvr\_dfd3.bp1', and 'analysis/dlvr\_idef3.bp1'. It also lists untracked files: '.gitignore' and 'hello.c'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

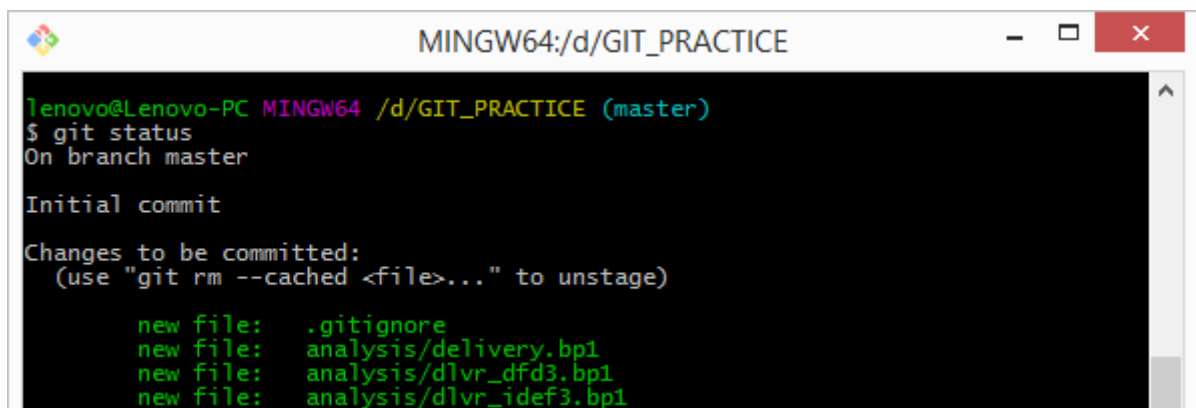
        new file:   analysis/delivery.bp1
        new file:   analysis/dlvr_dfd3.bp1
        new file:   analysis/dlvr_idef3.bp1

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

        .gitignore
        hello.c
```

Figure 1.13

As you can see, the previously created file `hello.c` is not tracked, because it is not in the list of files in the "Untracked files" section, and it was not added to the index. The `.gitignore` file must also be prepared for committing changes using the `git add` command (Figure 1.14):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the output of the 'git status' command after adding the '.gitignore' file. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The output shows that '.gitignore' is now a new file to be committed, along with the three files from the previous figure. 'hello.c' remains untracked.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   analysis/delivery.bp1
        new file:   analysis/dlvr_dfd3.bp1
        new file:   analysis/dlvr_idef3.bp1

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

        hello.c
```

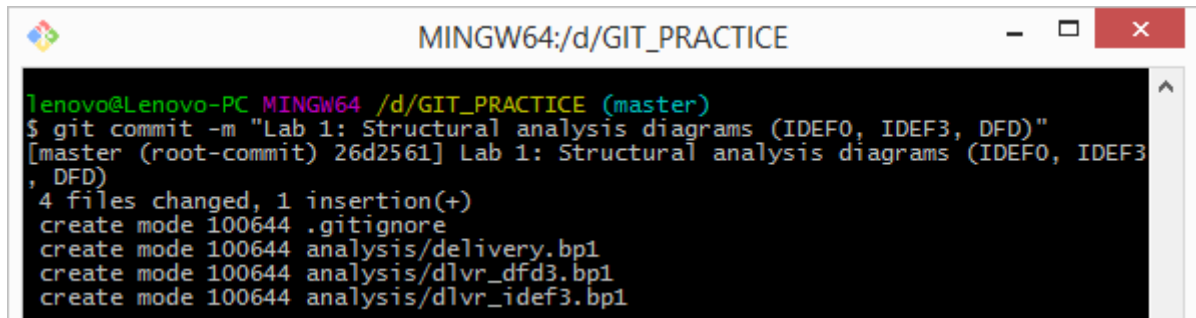
Figure 1.14

Now that the index is configured as needed, you can commit the changes you made. It is necessary to remember that everything that has not yet been indexed - any files created or changed by the user, and for which the `git add` command was not executed after the moment of editing - will not be included in



the commit. Such files will remain modified on the user's disk. In our case, it can be seen that everything is indexed (Figure 1.14) and ready for fixation.

The `git commit` command is used to commit changes. A comment on commits is usually typed on the command line along with the commit command, indicating after the `-m` option (Figure 1.15):

A screenshot of a terminal window titled "MINGW64:/d/GIT\_PRACTICE". The prompt is "lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)". The command entered is "\$ git commit -m 'Lab 1: Structural analysis diagrams (IDEF0, IDEF3, DFD)'". The output shows the commit message "[master (root-commit) 26d2561] Lab 1: Structural analysis diagrams (IDEF0, IDEF3, DFD)" and a summary of changes: "4 files changed, 1 insertion(+)", followed by a list of created files: ".gitignore", "analysis/delivery.bp1", "analysis/dlvr\_dfd3.bp1", and "analysis/dlvr\_idef3.bp1".

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git commit -m "Lab 1: Structural analysis diagrams (IDEF0, IDEF3, DFD)"
[master (root-commit) 26d2561] Lab 1: Structural analysis diagrams (IDEF0, IDEF3, DFD)
4 files changed, 1 insertion(+)
create mode 100644 .gitignore
create mode 100644 analysis/delivery.bp1
create mode 100644 analysis/dlvr_dfd3.bp1
create mode 100644 analysis/dlvr_idef3.bp1
```

Figure 1.15

Another way is to type `git commit` without parameters. This command will open a text editor with a default comment that contains the commented output of the `git status` command, which you can delete and type your own comment or leave as a reminder of what was committed.


After the commit was created, information was displayed about the branch to which the commit was made (master), the SHA-1 checksum of this commit (26d2561), how many files were changed (4 files changed), as well as statistics on added / removed lines in this commit (Figure 1.15).

In order to remove a file from Git, you need to remove it from the tracked files (more precisely, remove it from the index) and then commit. This will be done by the `git rm` command, which also removes the file from the working directory, so it will not be marked as untracked next time. If you simply delete the file from the working directory, it will be shown in the "Changes not staged for commit" section of the `git status` command output. And only after executing the `git rm` command, deleting the file will get into the index.

If the file has been modified and is already indexed, you must use forced deletion with the -f option.

A useful feature is to remove a file from the index while leaving it in the working directory. This is especially necessary when the user forgot to add something to the .gitignore file and mistakenly indexed, for example, a large log file or intermediate compilation files. For this, you need to use the --cached option.

Suppose that the DFD-diagram file needs to be removed from the index for some reason, but, at the same time, left available in the working directory for further editing or correction of detected errors (Figure 1.16):

A screenshot of a terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The terminal shows two commands and their outputs. The first command is '\$ git rm --cached analysis/dlvr\_dfd3.bp1', which results in 'rm 'analysis/dlvr\_dfd3.bp1''. The second command is '\$ git status', which shows the file 'analysis/dlvr\_dfd3.bp1' as deleted and untracked. The terminal text is as follows:

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git rm --cached analysis/dlvr_dfd3.bp1
rm 'analysis/dlvr_dfd3.bp1'

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    analysis/dlvr_dfd3.bp1

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

        analysis/dlvr_dfd3.bp1
```

Figure 1.16

As a result of executing the git rm command with the --cached option, in the next commit this file will be removed from the analysis directory in the repository, but will remain untracked by the Git system and available for editing in the working directory.

Fixing the deletion of the dlvr\_dfd3.bp1 file can be done in the second way (using the git commit command without parameters), by typing a comment to the commits in a text editor, having previously deleted the comment created by default (Figure 1.17):



Figure 1.17

By default, Git uses the convenient and concise Vim editor. To switch to editing mode, press `i` or `Insert`, to return to command mode, press `Esc`. You can save the file and exit the editor using the command: `wq` (for forced recording use: `wq!`), After which the output of the `git commit` command will be displayed (Figure 1.18):

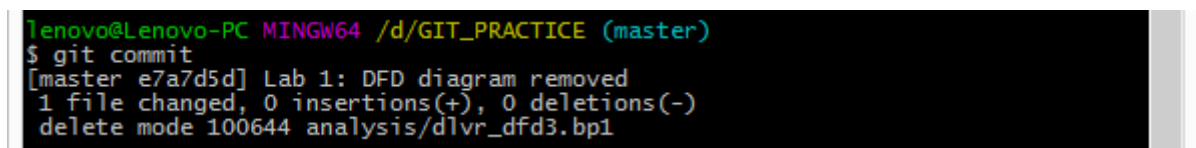
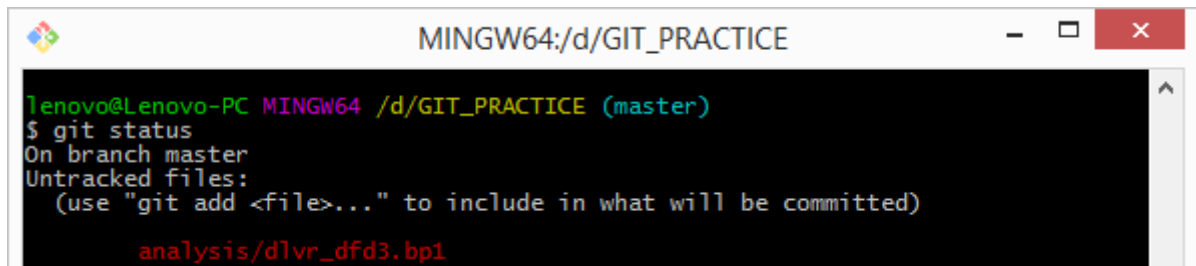


Figure 1.18

By typing the `git status` command, you can verify that the "remote" file is still available for editing in the working directory and is marked by the Git system as untracked (Figure 1.19):

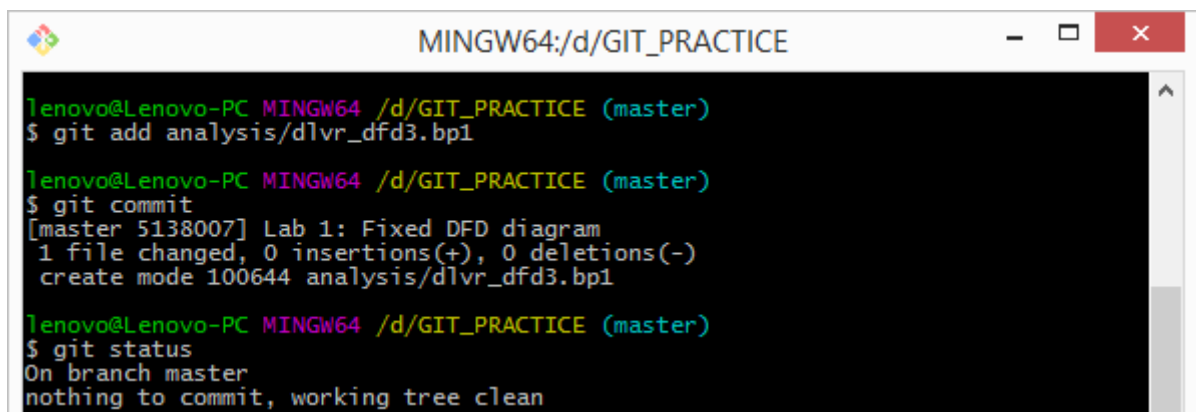
A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the output of the 'git status' command. The output indicates the current branch is 'master' and lists 'analysis/dlvr\_dfd3.bp1' as an untracked file.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

analysis/dlvr_dfd3.bp1
```

Figure 1.19

If you wish, you can make any changes to this file (for example, correct the shortcomings indicated by the teacher), index it, and fix the changes using the commit command (for convenience, you can use the Vim editor to enter a comment) as shown in Figure 1.20:

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing a sequence of Git commands: 'git add', 'git commit', and 'git status'. The commit message is 'Lab 1: Fixed DFD diagram'. The status shows the working tree is clean.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git add analysis/dlvr_dfd3.bp1

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git commit
[master 5138007] Lab 1: Fixed DFD diagram
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 analysis/dlvr_dfd3.bp1

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
nothing to commit, working tree clean
```

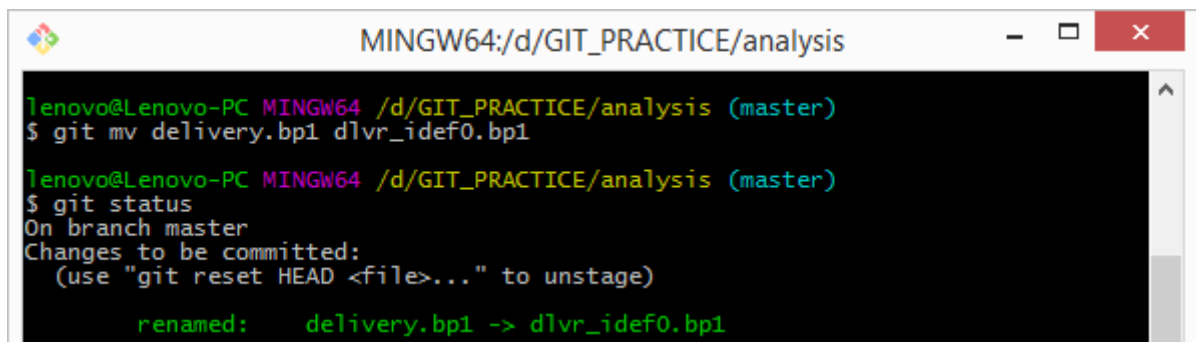
Figure 1.20

You can pass files, directories, or templates to the git rm command. For example, to remove all files with the extension .bp1 from the analysis directory, you can use the command `git rm analysis / \ *. bp1`.

Unlike many other version control systems, Git does not track file movements explicitly. When you rename a file in Git, it does not store any metadata to indicate that the file has been renamed.

If the file needs to be renamed or moved, use the git mv command.

For example, for one reason or another, you need to rename the file `delivery.bp1` to `dlvr_idef0.bp1` (Figure 1.21):



```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE/analysis (master)
$ git mv delivery.bp1 dlvr_idef0.bp1

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE/analysis (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    delivery.bp1 -> dlvr_idef0.bp1
```

Figure 1.21

After the `git mv` command is executed, if you check the status of the prepared files area with the `status` command, you will see that Git has indexed the file renaming (Figure 1.21).

However, this is equivalent to running the following commands:

```
mv delivery.bp1 dlvr_idef0.bp1
```

```
git rm delivery.bp1
```

```
git add dlvr_idef0.bp1
```

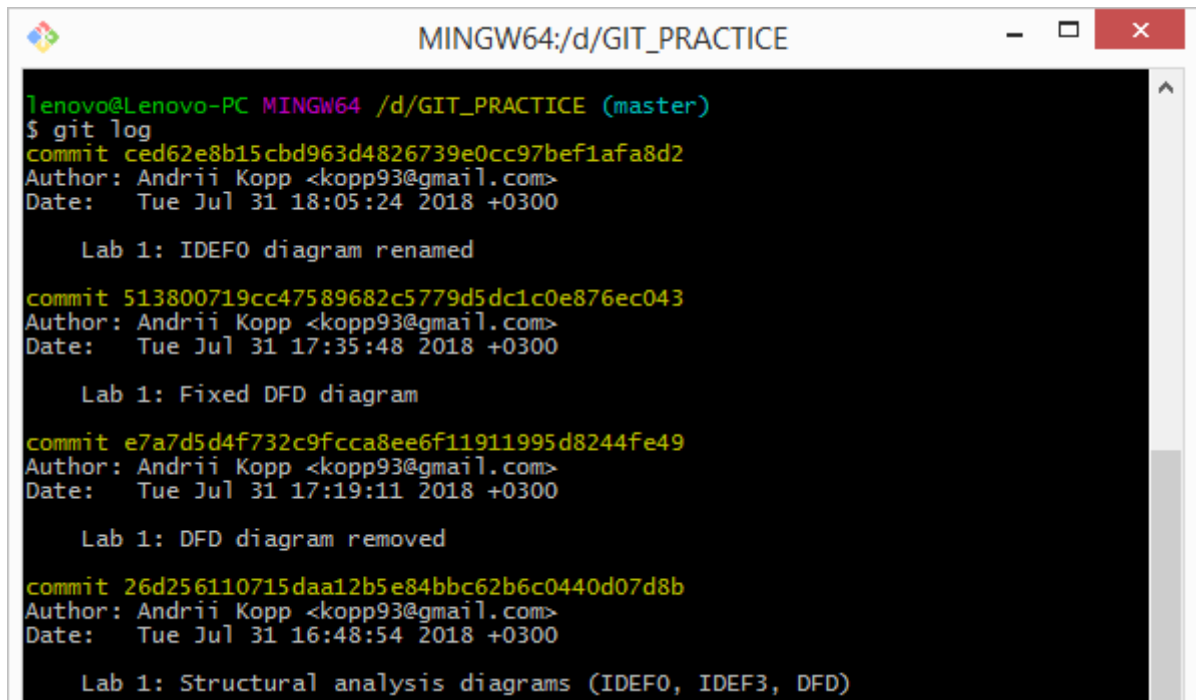
The Git system implicitly determines that a rename has occurred, so it doesn't matter which way the file is renamed.

After the file has been renamed, the changes must be committed again using the `git commit` command.

### 1.3.4 View commit history

To view the history of commits, a simple and at the same time powerful tool is used - the `git log` command. This command is especially useful when a user clones a repository with an existing commit history and wants to know what happened to that repository.

As a result of executing `git log`, a list of commits created in this repository will be displayed in reverse chronological order (Figure 1.22):



```
tenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git log
commit ced62e8b15cbd963d4826739e0cc97bef1afa8d2
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:05:24 2018 +0300

    Lab 1: IDEF0 diagram renamed

commit 513800719cc47589682c5779d5dc1c0e876ec043
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 17:35:48 2018 +0300

    Lab 1: Fixed DFD diagram

commit e7a7d5d4f732c9fcca8ee6f11911995d8244fe49
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 17:19:11 2018 +0300

    Lab 1: DFD diagram removed

commit 26d256110715daa12b5e84bbc62b6c0440d07d8b
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 16:48:54 2018 +0300

    Lab 1: Structural analysis diagrams (IDEF0, IDEF3, DFD)
```

Figure 1.22

One of the most useful options of the log command is -p, which shows the delta (difference) contributed by each commit. You can also use -2, which will limit the output to the last 2 records.

You can read more about the git log command at the link:

<https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>

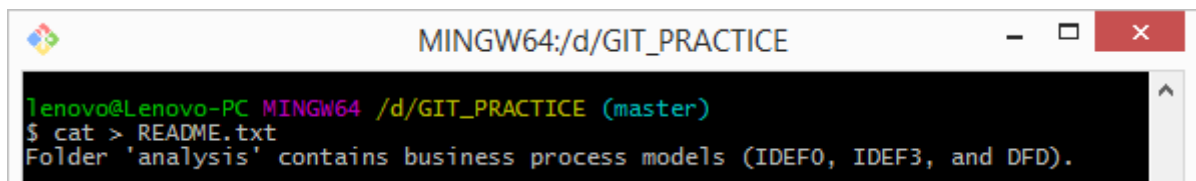
### 1.3.5 Cancellation of changes

At any stage of work, it may be necessary to cancel something. One of the few situations in Git where you can lose your work if you do something wrong is when it's not always possible to undo the undos themselves.

One of the typical exceptions occurs when the user commits too early, forgetting to add some files or entering the wrong commit comment. If you need to do this commit again, you can do git commit with the --amend option.

If there were no changes after the last commit, then the state of the project will be exactly the same and all that will need to be changed is the comment to the commits in the editor.

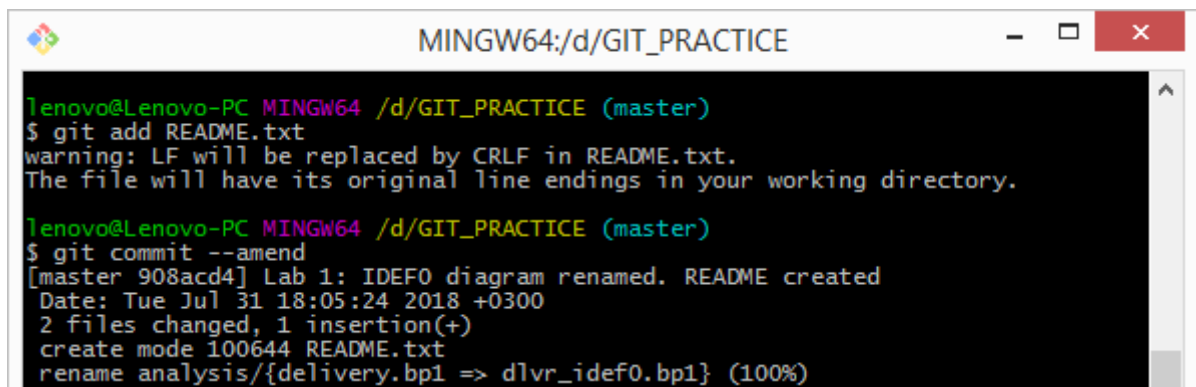
For example, before executing the last commit (fixing the renaming of the IDEF0 diagram file), it was also necessary to index the README.txt file with the following content (Figure 1.23):

A screenshot of a terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The command '\$ cat > README.txt' has been executed, and the output is 'Folder 'analysis' contains business process models (IDEF0, IDEF3, and DFD).'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ cat > README.txt
Folder 'analysis' contains business process models (IDEF0, IDEF3, and DFD).
```

Figure 1.23

Now, with the help of the following commands, you can add the README.txt file to the previous commit and edit the comment to the commits (Figure 1.24):


A screenshot of a terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The first command is '\$ git add README.txt', which outputs a warning: 'warning: LF will be replaced by CRLF in README.txt. The file will have its original line endings in your working directory.' The second command is '\$ git commit --amend', which outputs: '[master 908acd4] Lab 1: IDEF0 diagram renamed. README created', 'Date: Tue Jul 31 18:05:24 2018 +0300', '2 files changed, 1 insertion(+)', 'create mode 100644 README.txt', and 'rename analysis/{delivery.bp1 => dlvr\_idef0.bp1} (100%)'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git add README.txt
warning: LF will be replaced by CRLF in README.txt.
The file will have its original line endings in your working directory.

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git commit --amend
[master 908acd4] Lab 1: IDEF0 diagram renamed. README created
Date: Tue Jul 31 18:05:24 2018 +0300
2 files changed, 1 insertion(+)
create mode 100644 README.txt
rename analysis/{delivery.bp1 => dlvr_idef0.bp1} (100%)
```

Figure 1.24

You can check whether the last commit was really changed, and not a new one created, using the git log command (Figure 1.25):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the output of the 'git log' command. The output lists four commits with their hashes, authors, dates, and commit messages. The commit messages describe changes to IDEF0 diagrams and a README file.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git log
commit 908acd45edc20615c35c9d247aadddf482553c51
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:05:24 2018 +0300

    Lab 1: IDEF0 diagram renamed. README created

commit 513800719cc47589682c5779d5dc1c0e876ec043
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 17:35:48 2018 +0300

    Lab 1: Fixed DFD diagram

commit e7a7d5d4f732c9fcca8ee6f11911995d8244fe49
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 17:19:11 2018 +0300

    Lab 1: DFD diagram removed

commit 26d256110715daa12b5e84bbc62b6c0440d07d8b
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 16:48:54 2018 +0300

    Lab 1: Structural analysis diagrams (IDEF0, IDEF3, DFD)
```

Figure 1.25

To demonstrate how to unindex a file, it is necessary to create two files `index.html` and `script.js` in the web directory (Figure 1.26):

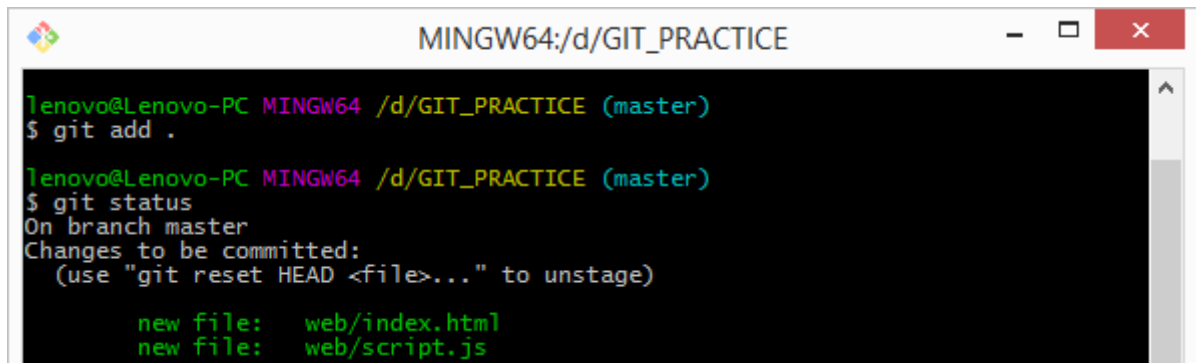
A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the contents of two files. First, 'cat web/index.html' is run, displaying an HTML document with a title 'Index' and a script tag. Then, 'cat web/script.js' is run, displaying a JavaScript snippet that sets the innerHTML of a div with id 'hello' to 'Hello World!'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ cat web/index.html
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<div id="hello"></div>
<script src="script.js"></script>
</body>
</html>
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ cat web/script.js
var layout = document.getElementById("hello");
layout.innerHTML = "Hello World!";
```

Figure 1.26

To index all untracked project files, you can use the `git add` command in Git. (Figure 1.27):



A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the execution of 'git add .' followed by 'git status'. The status output indicates two new files are staged for commit: 'web/index.html' and 'web/script.js'.

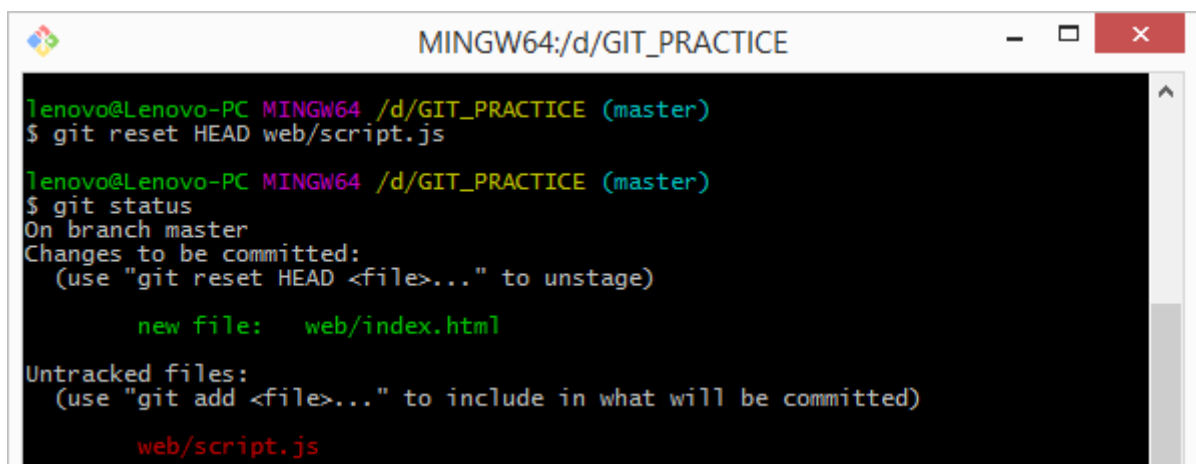
```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git add .

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   web/index.html
        new file:   web/script.js
```

Figure 1.27

But what if these two files had to be written in two separate commits? The conclusion of the git status command suggests that you can cancel the indexing of one of the two files using the git reset command (Figure 1.28):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the execution of 'git reset HEAD web/script.js' followed by 'git status'. The status output shows 'web/index.html' as a new file to be committed, while 'web/script.js' is listed as an untracked file.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git reset HEAD web/script.js

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

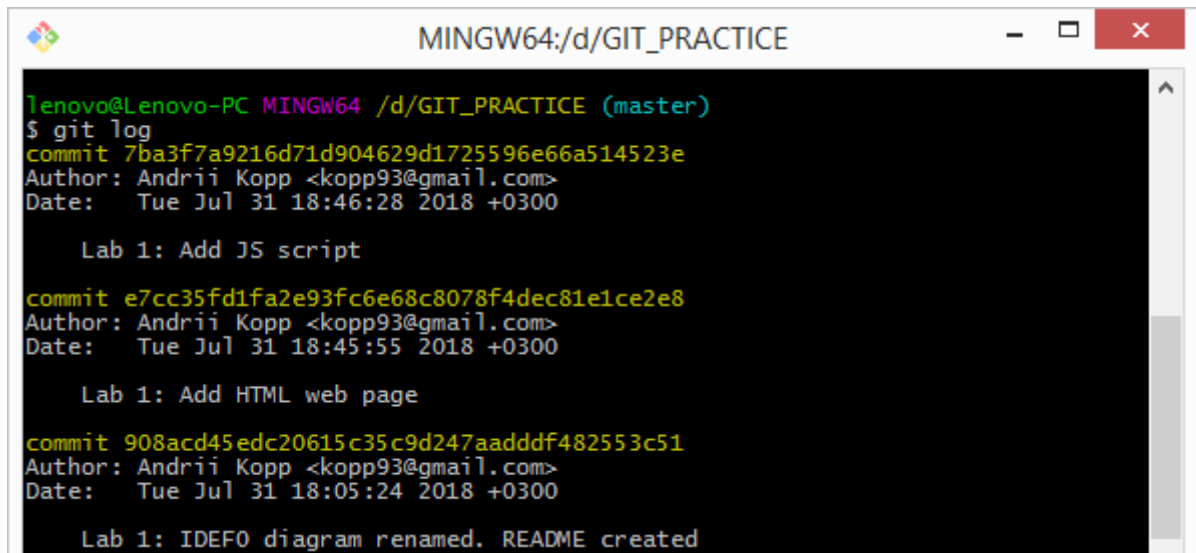
        new file:   web/index.html

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

        web/script.js
```

Figure 1.28

Now that the second created file (script.js) is considered not indexed, you can commit the first file (index.html), add the second file to the prepared files area, and commit it in a separate commit (Figure 1.29).

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the output of the 'git log' command. The output lists three commits with their hashes, authors, dates, and descriptions. The descriptions are 'Lab 1: Add JS script', 'Lab 1: Add HTML web page', and 'Lab 1: IDEFO diagram renamed. README created'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git log
commit 7ba3f7a9216d71d904629d1725596e66a514523e
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:46:28 2018 +0300

    Lab 1: Add JS script

commit e7cc35fd1fa2e93fc6e68c8078f4dec81e1ce2e8
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:45:55 2018 +0300

    Lab 1: Add HTML web page

commit 908acd45edc20615c35c9d247aadddf482553c51
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:05:24 2018 +0300

    Lab 1: IDEFO diagram renamed. README created
```

Figure 1.29

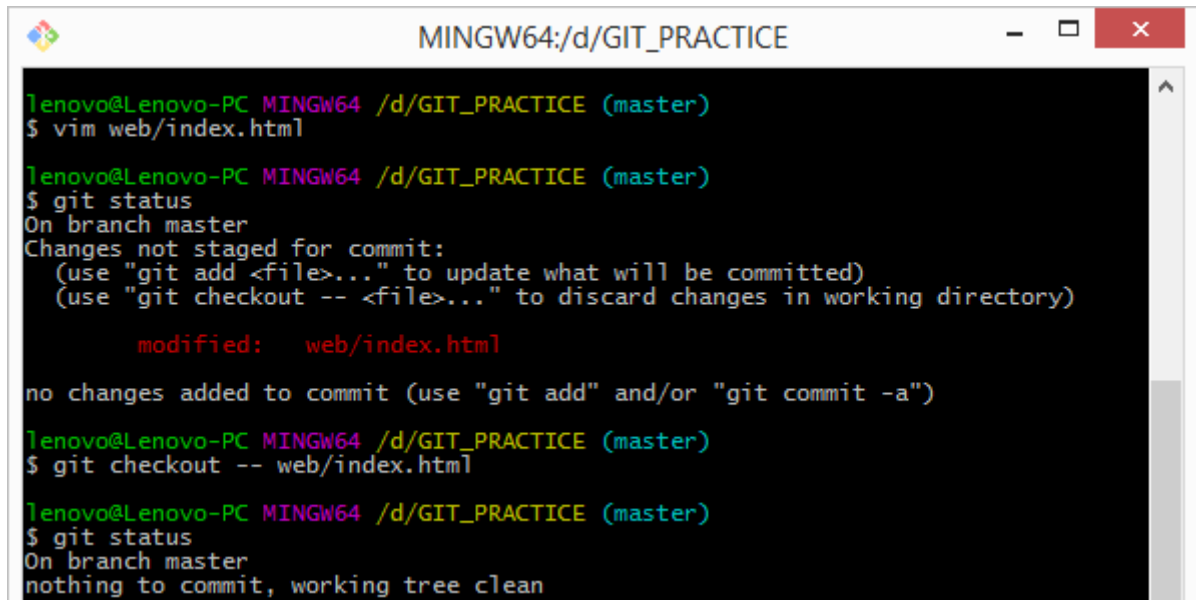
Let's say you want to display the text "Hello World" on the "Index" web page as a first-level header. To do this, we will make appropriate changes to the index.html file (Figure 1.30):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the content of the 'index.html' file. The content is an HTML document with a title 'Index' and a body containing a header 'Hello World' and a script tag for 'script.js'. The status bar at the bottom shows the file path, encoding, date, and line numbers.

```
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
    <h1><div id="hello"></div></h1>
<script src="script.js"></script>
</body>
</html>
```

Figure 1.30

But now it turned out that it is not necessary to leave these changes. To quickly cancel changes, return the file to the state it was in during the last commit, you can use the git checkout command (Figure 1.31):

A screenshot of a Windows terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The terminal shows a series of commands and their outputs. First, 'vim web/index.html' is run. Then, 'git status' is run, showing that 'web/index.html' is modified. Next, 'git checkout -- web/index.html' is run. Finally, 'git status' is run again, showing that the working tree is clean.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ vim web/index.html

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   web/index.html

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

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git checkout -- web/index.html

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Figure 1.31

To verify that the file has returned to the state in which it was at the time of the last commit, it is necessary to review its contents (Figure 1.32):

A screenshot of a Windows terminal window titled 'MINGW64:/d/GIT\_PRACTICE'. The terminal shows the command 'cat web/index.html' being run, which displays the contents of the file. The content is an HTML document with a title 'Index' and a script tag.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ cat web/index.html
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<div id="hello"></div>
<script src="script.js"></script>
</body>
</html>
```

Figure 1.32

It is necessary to understand that git checkout is a dangerous command: all the changes made in this file are lost - another file was simply copied over it.

You should never use this command unless you are absolutely certain that the file is not needed. Stash and branching are more preferred methods. These methods will be discussed later.

### **1.5 Questions for self-testing**

1. What is the main difference between Git and other version control systems?

2. What are file system snapshots in Git and what are they for?

3. What is the peculiarity of storing project history in the Git system? The main advantages and disadvantages of this system?

4. What is a commit (fixation of changes in the project) and what is it for?

5. How does the Git system control data integrity? What is a SHA-1 hash?

6. How does the Git system record the actions performed by the user? What is the advantage of this way of organizing changes?

7. In what states can files be in the Git system? Briefly describe each of these states.

8. What file storage areas exist in projects using Git as a version control system?

9. Describe the main stages of the workflow using the Git version control system. In what cases is a file considered modified, prepared or committed?

10. How is the initial configuration of the Git system carried out? Purpose of the git config command.

11. What is the --global option of the git config command for?

12. What are the ways to create a repository in Git? Purpose of the git init command.

13. How do tracked files in the working directory differ from untracked files?

14. What is the git status command for? What information is displayed when this command is executed?

15. In which branch does the Git system work by default?
16. What is the git add command for? What are the main features of using this command?
17. How can you avoid the indexing of unwanted files (logs, program compilation results, etc.)?
18. What is the git commit command for? What are the main features of using this command?
19. What information will be displayed as a result of executing the git commit command?
20. How can I delete a file from Git? What to do if the file has already been indexed? How do I remove a file from the index but leave it in the working directory?
21. Which command can be used to delete all files from the working directory with a certain extension?
22. How are files moved or renamed in the Git system? What command is used for this? What are the alternative ways of performing these operations?
23. What is the git log command for? What parameters of this command can be used to display more detailed information?
24. How in the Git system can you redo the last commit, taking into account the necessary changes? Under what conditions is such an operation possible?
25. How can a file be unindexed? How to index all untracked files?
26. What is the git checkout command for? Why should this command be used with caution?

## **2.4 Working with branches in the Git system**

### **2.4.1 Branching and merging**

As a result, there are already several commits in the master branch used by the Git system by default (Figure 2.12).

```
MINGW64:/d/GIT_PRACTICE
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git log
commit d53a9ee238e95d8cf89d2ab6a056550d718dd020
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 11:55:57 2018 +0300

    Lab 2: UML models

commit ea3206b1343ffd67916f7edd6387790ab1926c95
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 11:54:42 2018 +0300

    Lab 2: User stories list

commit 7ba3f7a9216d71d904629d1725596e66a514523e
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:46:28 2018 +0300

    Lab 1: Add JS script

commit e7cc35fd1fa2e93fc6e68c8078f4dec81e1ce2e8
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:45:55 2018 +0300
```

Figure 2.12

Let's imagine that after documenting the requirements and designing the architecture of the system being created, it is necessary to move on to work on creating a prototype of the information system.

Since prototyping is a separate task within the traditional software development lifecycle (Figure 2.13), it is necessary to create a new branch in the Git version control system and work on it.

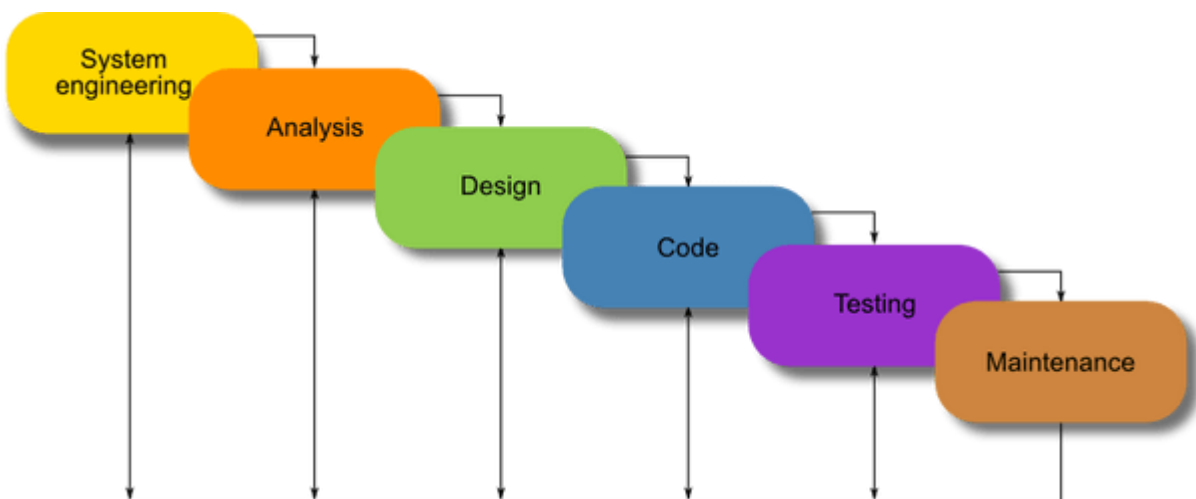
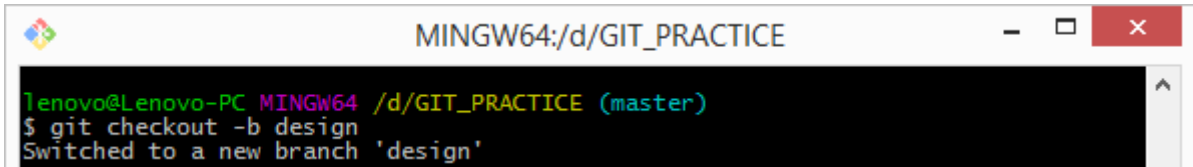


Figure 2.13

To create a new branch and switch to it, you need to use the git checkout command with the -b key (Figure 2.14):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' with standard window controls. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)'. The command '\$ git checkout -b design' has been entered, and the output is 'Switched to a new branch 'design''.

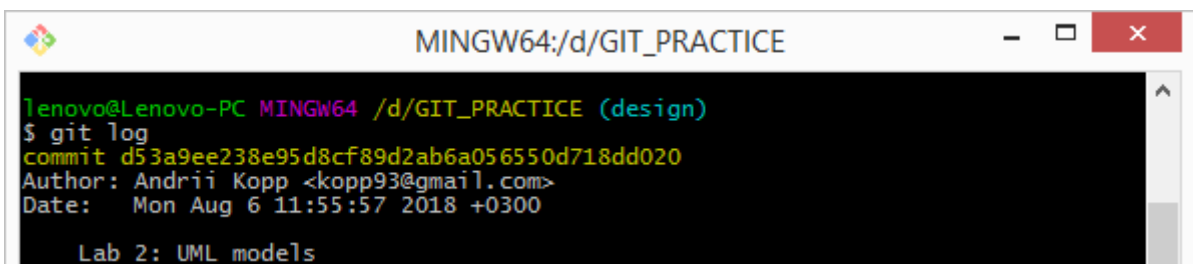
```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git checkout -b design
Switched to a new branch 'design'
```

Figure 2.14

Running the checkout command with the -b switch is a shortcut for two commands:

```
git branch design # creating a new branch
git checkout design # switch to new branch
```

After creation, the new branch points to the same commit as the master branch, since no changes have been committed to the design branch yet (Figure 2.15).

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' with standard window controls. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (design)'. The command '\$ git log' has been entered, and the output shows a single commit with its hash, author, date, and file path.

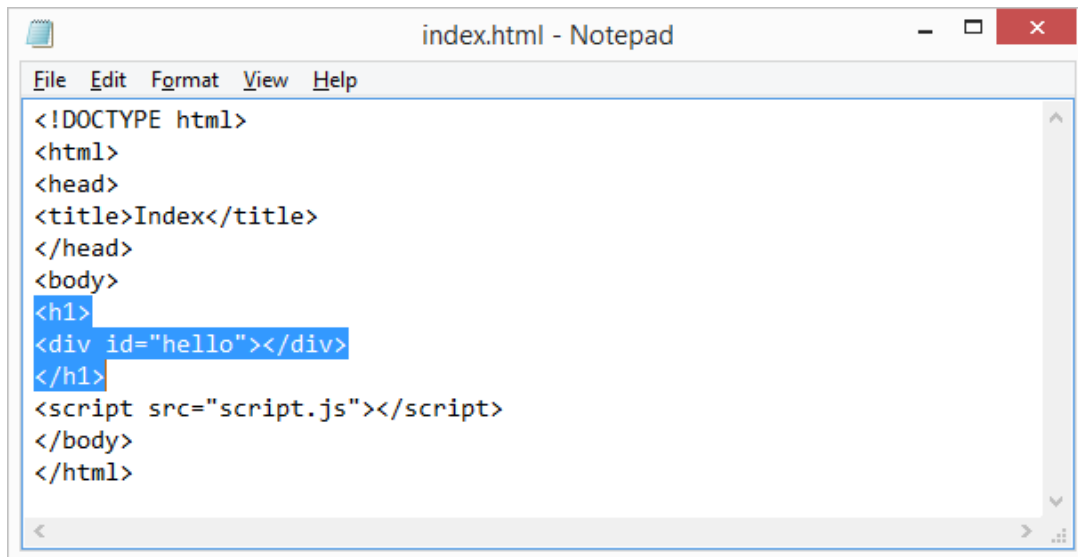
```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)
$ git log
commit d53a9ee238e95d8cf89d2ab6a056550d718dd020
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 11:55:57 2018 +0300

    Lab 2: UML models
```

Figure 2.15

Of course, during the work on the prototype of the created system, some changes will be made and fixed. For example, it became necessary to use the

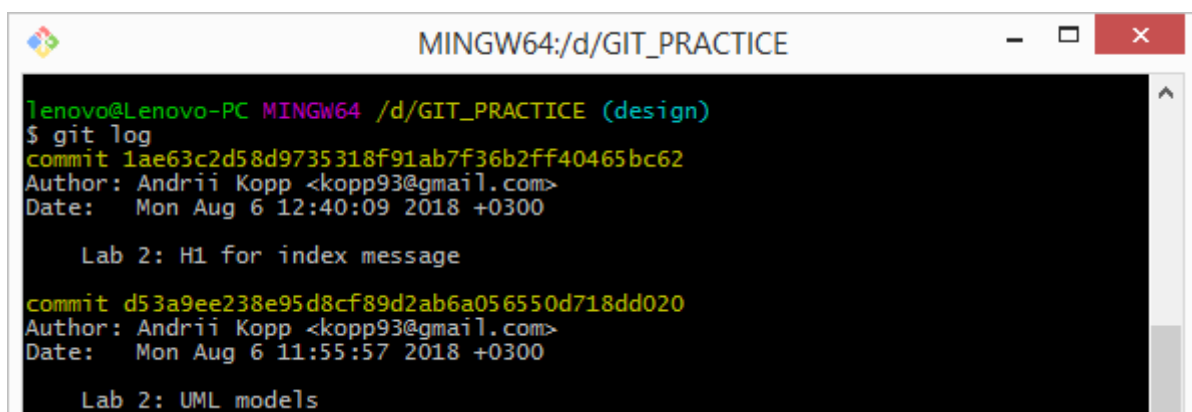
first-level header for the message displayed on the index.html page (Figure 2.16):



```
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<h1>
<div id="hello"></div>
</h1>
<script src="script.js"></script>
</body>
</html>
```

Figure 2.16

The changes made must be recorded. After the commits, the design branch in which the work was performed will point to the last commit associated with adding the header to the index.html file, that is, it will move forward relative to the master branch (Figure 2.17).



```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)
$ git log
commit 1ae63c2d58d9735318f91ab7f36b2ff40465bc62
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 12:40:09 2018 +0300

    Lab 2: H1 for index message

commit d53a9ee238e95d8cf89d2ab6a056550d718dd020
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 11:55:57 2018 +0300

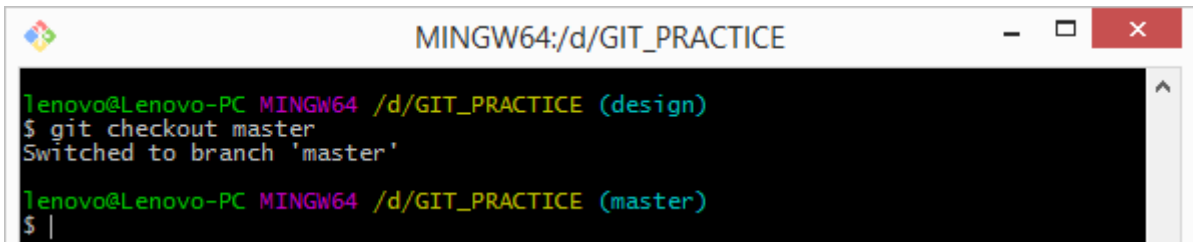
    Lab 2: UML models
```

Figure 2.17



Suppose that for some reason it was necessary to change the color of the displayed message on the index.html page to red. Moreover, it is necessary to do this in the following way:

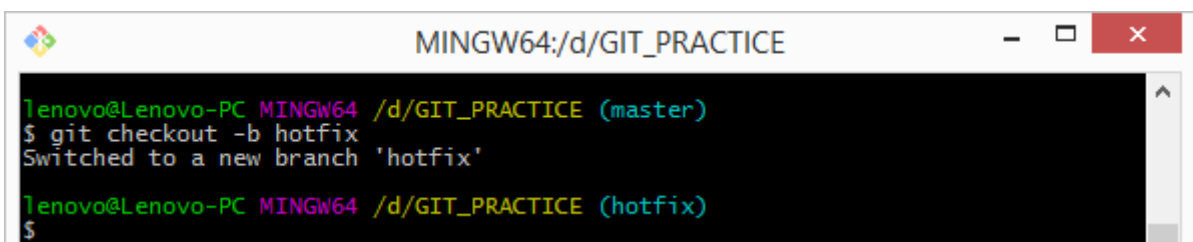
1. After making sure that all changes in the design branch are recorded, switch to the master branch (Figure 2.18):

A screenshot of a Windows terminal window titled "MINGW64:/d/GIT\_PRACTICE". The prompt is "lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (design)". The user enters the command "\$ git checkout master", and the terminal responds with "Switched to branch 'master'". The prompt then changes to "lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)".

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)
$ git checkout master
Switched to branch 'master'
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ |
```

Figure 2.18

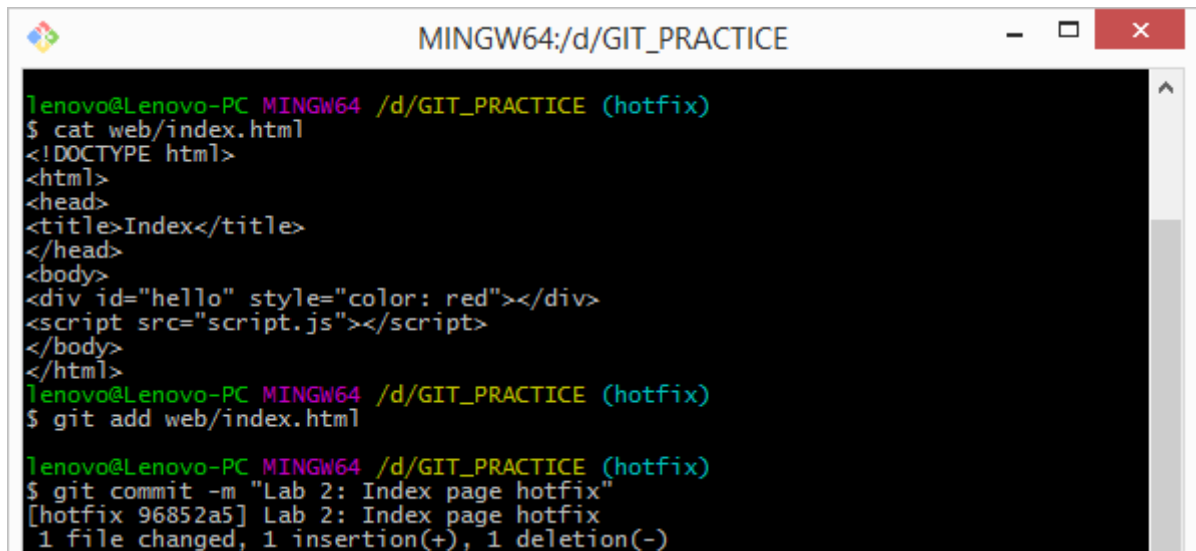
2. Create a branch in which the work will be performed (Figure 2.19):

A screenshot of a Windows terminal window titled "MINGW64:/d/GIT\_PRACTICE". The prompt is "lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (master)". The user enters the command "\$ git checkout -b hotfix", and the terminal responds with "Switched to a new branch 'hotfix'". The prompt then changes to "lenovo@Lenovo-PC MINGW64 /d/GIT\_PRACTICE (hotfix)".

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$
```

Figure 2.19

3. Make the necessary changes to the index.html file and make a commit with the appropriate comment (Figure 2.20):



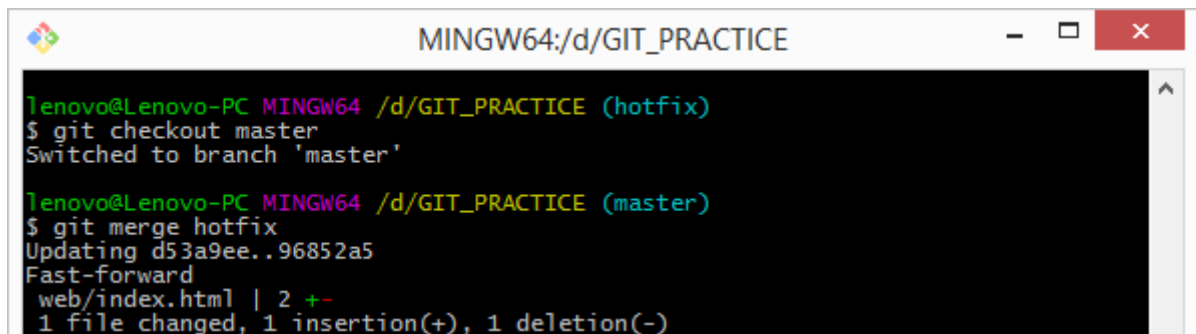
```
MINGW64:/d/GIT_PRACTICE

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ cat web/index.html
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<div id="hello" style="color: red"></div>
<script src="script.js"></script>
</body>
</html>
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ git add web/index.html

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ git commit -m "Lab 2: Index page hotfix"
[hotfix 96852a5] Lab 2: Index page hotfix
1 file changed, 1 insertion(+), 1 deletion(-)
```

Figure 2.20

4. Merge changes into the master branch to include them in the project using the git merge command (Figure 2.21):



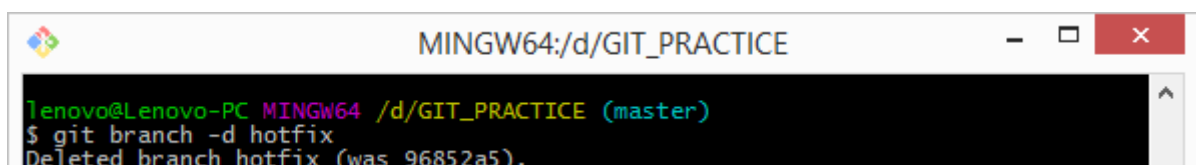
```
MINGW64:/d/GIT_PRACTICE

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ git checkout master
Switched to branch 'master'

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git merge hotfix
Updating d53a9ee..96852a5
Fast-forward
 web/index.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Figure 2.21

5. Remove the hotfix branch that is no longer needed (the master branch after the merge points to the same place, since the Git system simply moved the indicator forward due to the absence of divergent changes that would need to be merged together) using the git branch command with the -d option (Figure 2.22):



```
MINGW64:/d/GIT_PRACTICE

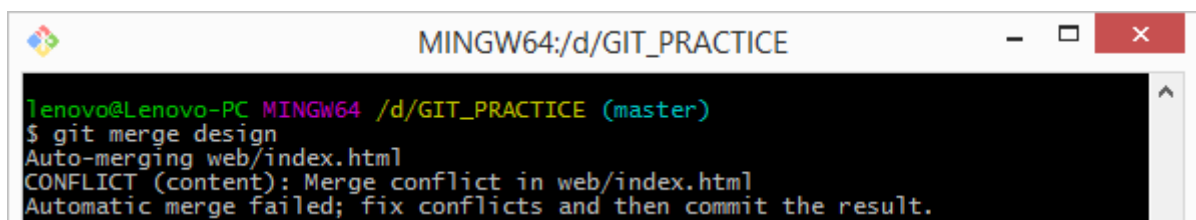
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git branch -d hotfix
Deleted branch hotfix (was 96852a5).
```

Figure 2.22

After the problem is solved, you can go back to the design branch and continue working. However, it is necessary to remember that the work done on the hotfix branch is not included in the commits on the design branch. If necessary, the master branch can be merged into the design branch using the `git merge master` command. In addition, the integration of changes can be delayed until the changes on the design branch have NOT been decided to be included in the main branch of the master project.

### 2.4.2 Merger conflicts

The process of merging branches does not always go smoothly. In our case, the problem solution in the design branch modifies the same file (`index.html`) as the hotfix branch, resulting in a merge conflict (Figure 2.23):

A screenshot of a terminal window titled "MINGW64:/d/GIT\_PRACTICE". The terminal shows the following text: 

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git merge design
Auto-merging web/index.html
CONFLICT (content): Merge conflict in web/index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Figure 2.23

The Git system will not create a new commit to merge branches, but will pause the process until the user resolves the conflict. To view files that have not been merged, it is necessary to execute the `git status` command (Figure 2.24):

A terminal window titled 'MINGW64:/d/GIT\_PRACTICE' showing the output of the 'git status' command. The output indicates that the user is on the 'master' branch and has unmerged paths. Specifically, it lists 'web/index.html' as a file where both the working directory and the index have modifications. The terminal text is as follows:

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

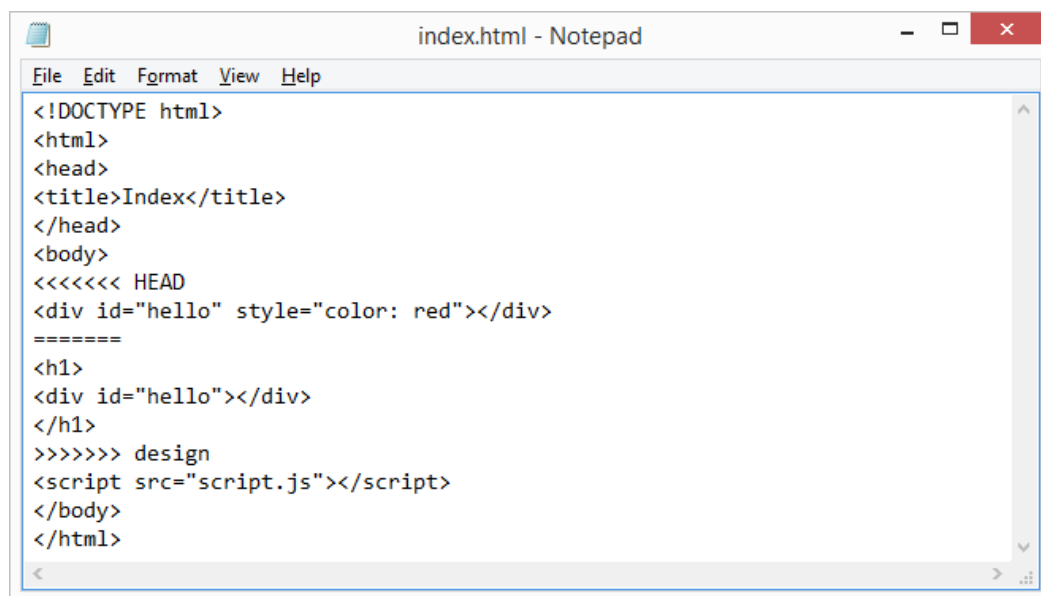
Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   web/index.html

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

Figure 2.24

The Git system adds standard markers to files (here it is index.html) that have a conflict (unmerged), so that you can open such files manually and resolve these conflicts. The index.html file will look like this (Figure 2.25):

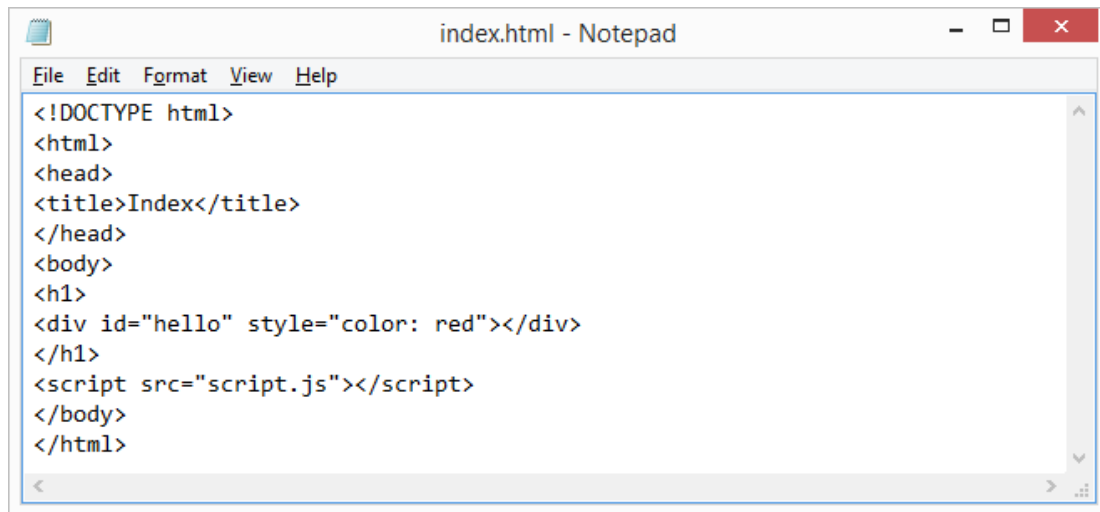
A Notepad window titled 'index.html - Notepad' showing the content of the 'index.html' file. The file contains HTML code with conflict markers. The code is as follows:

```
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<<<<<<< HEAD
<div id="hello" style="color: red"></div>
=====
<h1>
<div id="hello"></div>
</h1>
>>>>>> design
<script src="script.js"></script>
</body>
</html>
```

Figure 2.25

In the index.html file, everything above ===== is the version from HEAD (the master branch, since it was on it that the merge command was executed). Everything below is the version in the design branch. To resolve the conflict, it is necessary to either choose one of these parts, or somehow combine

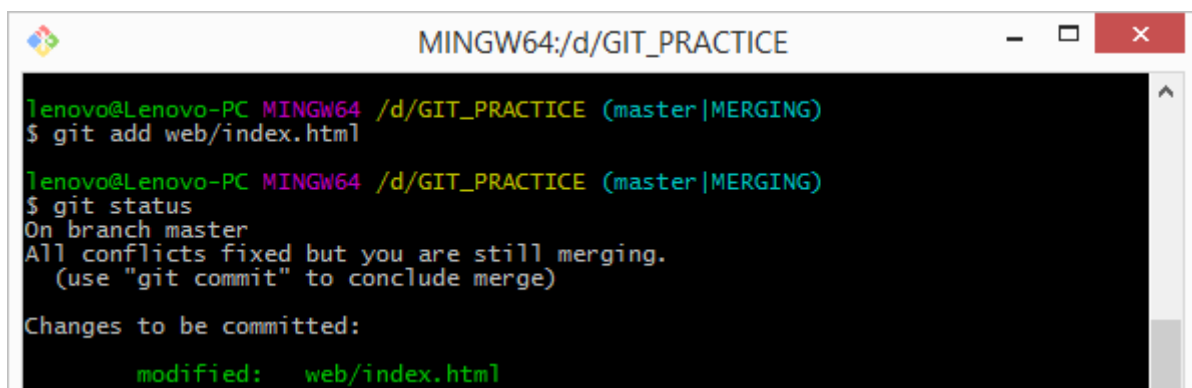
the contents at your discretion. For example, the conflict can be resolved as follows (Figure 2.25):



```
File Edit Format View Help
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<h1>
<div id="hello" style="color: red"></div>
</h1>
<script src="script.js"></script>
</body>
</html>
```

Figure 2.25

After resolving the conflicts, it is necessary to execute the git add command for each conflicting file. Indexing for the Git system will mean that all conflicts are allowed (Figure 2.26):



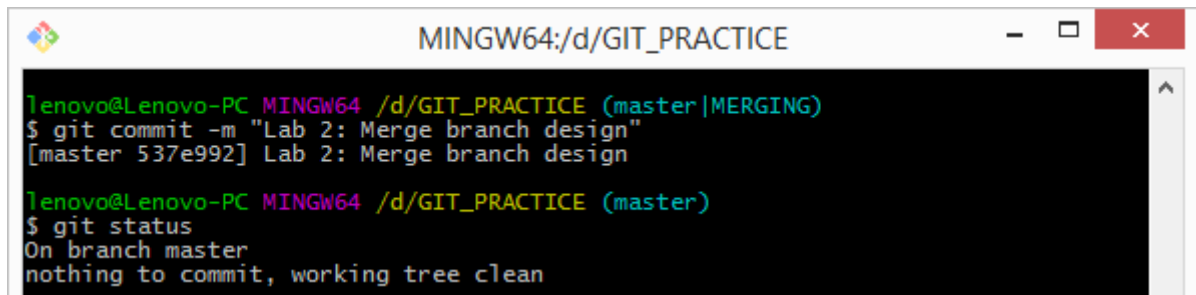
```
MINGW64:/d/GIT_PRACTICE
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master|MERGING)
$ git add web/index.html

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master|MERGING)
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:
  modified:   web/index.html
```

Figure 2.26

After making sure that all the files that had conflicts have been indexed, you can perform a git commit (Figure 2.27):



```
tenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master|MERGING)
$ git commit -m "Lab 2: Merge branch design"
[master 537e992] Lab 2: Merge branch design

tenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Figure 2.27

It is recommended to include information about how the conflict was resolved in the commit comment, unless it is obvious and may be useful for other users in the future.

## 2.6 Questions for self-testing

15. Which command can be used to create a new branch in the Git system?
16. What command in the Git system is used to switch between branches?
17. What is the -b key of the git checkout command used for?
18. What is the git merge command used for?
19. How can you delete a branch in Git?
20. For what reasons can conflicts arise when merging branches in the Git system?
21. How can you view the list of files that have not been merged?
22. How does the Git system "help" resolve conflicts in unmerged files?
23. What must be done to complete the branch merge process after all conflicts have been resolved?
24. How to avoid conflicts when merging branches?