МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИСТиД (филиал) в г.Пятигорске

Методические указания по выполнению лабораторных работ

По дисциплине «Проектирование, внедрение, сопровождение, настройка и эксплуатация информационных систем»

Направление подготовки	09.03.02 Информационные системы и технологии
Профиль	**Информационные системы и технологии**
Квалификация выпускника	Бакалавр
Форма обучения	очный
Учебный план	2020
Изучается в 7 семестре	

Пятигорск, 2020 г.

Рассмотрено и утверждено на заседании кафедры систем управления и информационных технологий протокол № ___ от _______________2020

Зав.кафедрой СУИТ __________________________________И.М. Першин

Введение

В лабораторный практикум по дисциплине «Методы и средства проектирования информационных систем и технологий» включены лабораторные работы по основным разделам этой дисциплины, читаемой на кафедре «Систем управления и информационных технологий». Лабораторные работы ориентированы на приобретение навыков студентами построения геометрических двух - и трехмерных объектов, сложных полигонов, позволяющим создавать реалистические изображения на экране персонального компьютера, а также на создание графических изображений и оформление проектно-конструкторской документации в современных системах автоматизированного проектирования. Работы ориентированы на использование систем контроля версий GitHub и средств проектирования Rational Rose.

Содержащиеся в практикуме сведения теории, методические указания и рекомендации по выполнению лабораторных работ позволяют использовать его в качестве дополнительного пособия для закрепления курса лекций.

Целью данного лабораторного практикума является поэтапное формирование у студентов знаний, умений и навыков работы с современными системами контроля версий, а также уметь работать с CASE-средством Rational Rose. Практикум предназначен для студентов Северо-Кавказского федерального университета и может быть полезным для всех желающих ознакомиться с основами автоматизированного проектирования.

Данный вид работы играет важную роль в формировании практических навыков работы с графической информацией и способствует формированию следующих образовательных компетенций:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

СОДЕРЖАНИЕ

Введение.. 3

Лабораторная работа №1. Система контроля версий Git.. 6

Лабораторная работа №2 . Создание Git-репозитория. 9

Лабораторная работа №3. Просмотр истории коммитов. Операции отмены. 14

Лабораторная работа №4. Работа с удалёнными репозиториями 23

Лабораторная работа №5. Работа с метками….. 28

Лабораторная работа №6. Работа с ветками 33

Лабораторная работа №7. Работа с GitHub. 51

Лабораторная работа №8. Знакомство с Rational Rose 63

Лабораторная работа №9. Диаграмма претендентов 66

Лабораторная работа №10. Отношения на диаграммах 67

Лабораторная работа №11. Диаграмма прецедентов проекта 70

Лабораторная работа №12. Диаграмма сценариев поведения... 74

Лабораторная работа №13. Диаграмма классов концептуальной схемы базы данных 77

Лабораторная работа №14. Установка связей между классами 80

Лабораторная работа №15. Моделирование программных систем.................................. 84

Лабораторная работа №16. Логическая схема базы данных 88

Лабораторная работа №17. Диаграмма деятельности 90

Лабораторная работа №18. Диаграмма классов... 95

Лабораторная работа №19. Генерация программного кода.. 100

Лабораторная работа №1. Система контроля версий Git.

Цель работы:

Изучить систему контроля версий Git, для чего она нужна и как установить на компьютер.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

В настоящее время владение Git стало обязательным требованием при приёме на работу не только для профессионалов, но даже для стажеров.

Git– это система контроля версий (СКВ). Существует несколько подобных систем, однако Git – на настоящий момент наиболее используемая СКВ.

Git создан для решения нескольких проблем любого программиста.

1) История изменений. Работа программиста – это всегда история изменений в коде программы.

a. Внося изменения в файлы, хочется знать ответ на вопросы «Кто сделал? Что сделал? Когда сделал?». Таким образом легко отслеживать, когда появились ошибки и кто их сделал. **Любая система, которая позволит нам видеть такую историю изменений, и является системой контроля версий.**

b. Иметь возможность отмены изменений или отката по истории назад (и вперёд).

2) Лёгкость внесения изменений.

a. Если нужно попробовать какой-то вариант – это должно быть просто.

b. Вернуться на основной вариант – также просто

c. Возможность легко принять или отвергнуть альтернативу.

3) Совместная работа

a. Хорошо тому живется, у кого одна нога… Если ног несколько, возникает проблема синхронизации.

b. Изменения разных пользователей нужно изолировать друг от друга…

c. … а по мере готовности – сливать вместе.

Программа Git решает все эти проблемы. Эта программа – набор скриптов, который умеет управлять изменениями. Он следит за файлами, ведет их историю, умеет ими манипулировать, откатывать, сливать и т.д.

Git используется в разработке ядра Linux и создан Линусом Торвальдсом.

Задача Git – вести полную историю изменений в некоей папке на сервере или локальном компьютере (**репозиторий**). К изменениям относятся добавление и удаление файлов, модификация их содержимого (нужно также учитывать, что Git не следит за пустыми папками, в папке нужно создать хотя бы один файл).

В историю также записывается, кто и когда сделал изменения.

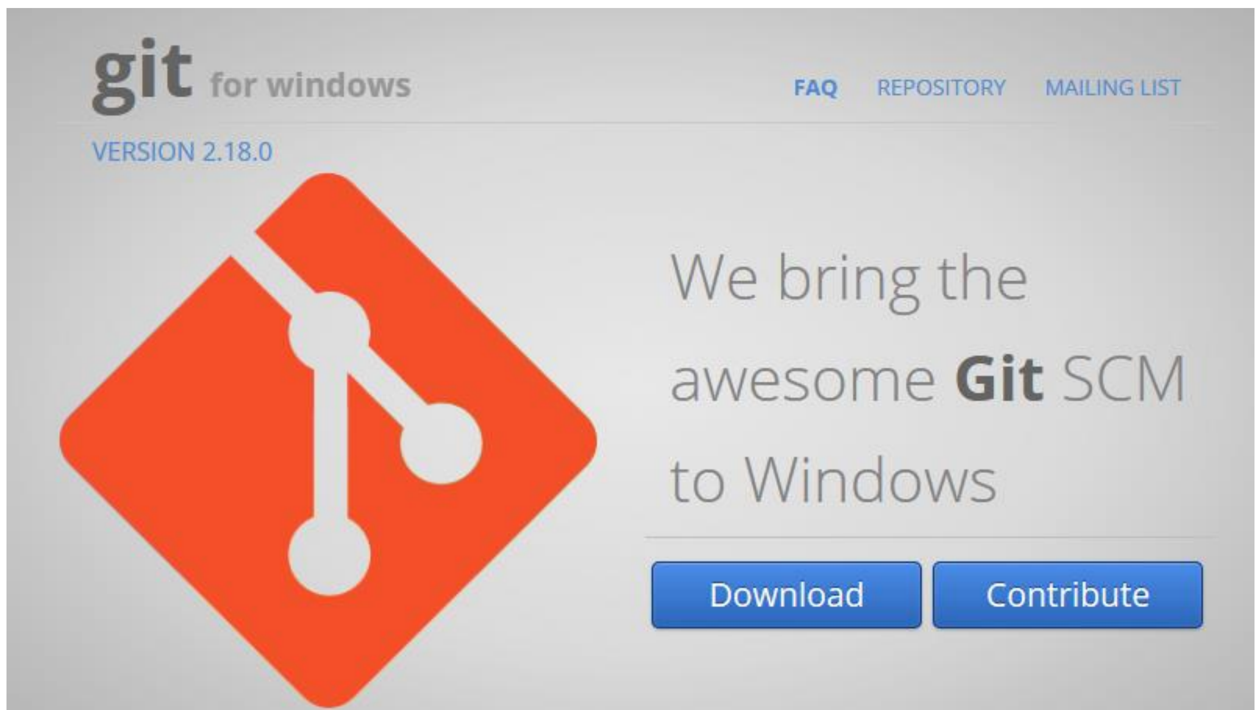
Git – это распределенная система контроля версий, в ней нет центрального репозитория. Репозиторий – это просто папка с вашими файлами, в которой есть ещё некая служебная информация для Gita. Их может быть очень много. Репозитории могут обмениваться изменениями между собой. Однако, это не DropBox, он не занимается хранением файлов, его задача следующая. Он может от одного репозитория к другому передать изменения!

Практическая часть.

1. Установка Git.

Для Windowsдля установки нужно перейти по ссылке.

git-for-windows.github.io



Данная версия программы представляет собой не только Git, но и нужное для его работы окружение, которое установится одним пакетом.

Нажимаем Download(вторая кнопка позволяет принять участие в разработке и нам пока не нужна ☺). Устанавливаем всё с настройками по умолчанию.

На компьютерах в лаборатории уже установлен Git, можно переходить к следующей части.

Контрольные вопросы:

1. Что такое СКВ?
2. Какие проблемы решает Git?
3. Для разработки какой операционной системы используется Git?
4. Что такое репозиторий?
5. В чем разница между GoogleDrive/DropBox/YandexDisk и репозиторием Git?

Список литературы:

1. Михеева, Е. В. Информационные технологии в профессиональной деятельности :учеб.пособие / Е.В. Михеева. - 14-е изд., стер. - М. : Академия, 2016. - 384 с.
2. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.

Лабораторная работа №2 . Создание Git-репозитория.

Цель работы:

Научиться создавать новые репозитории в системе контроля Git в среде GitBash. Научиться правильно перемещаться внутри проекта с использованием команд в командной строке, а также установка авторства для проекта, с целью отслеживания изменений.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Система спроектирована как набор программ, специально разработанных с учётом их использования в сценариях. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы. Например, Cogito является именно таким примером оболочки к репозиториям Git, а StGit использует Git для управления коллекцией исправлений (патчей).

Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Как и Darcs, BitKeeper, Mercurial, Bazaar и Monotone[en], Git предоставляет каждому разработчику локальную копию всей истории разработки, изменения копируются из одного репозитория в другой.

Удалённый доступ к репозиториям Git обеспечивается git-демоном, SSH- или HTTP-сервером. TCP-сервис git-daemon входит в дистрибутив Git и является наряду с SSH наиболее распространённым и надёжным методом доступа. Метод доступа по HTTP, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров.

Практическая часть.

Создание репозитория.

2. Запустите GitBash из меню Пуск - Git.

Откроется следующая консоль:

```
MINGW64:/c/Users/Admin
Admin@File-Server MINGW64 ~
$ |
```

Это командная строка Linux (наподобие консоли командной строки Windows), аккуратно перенесенная в Windows.

Далее работа с Git будет объясняться на примере работы с консольным клиентом по следующим причинам:

- Чтобы у вас складывалось понимание происходящего и при возникновении проблем вы могли четко объяснить, что вы делали, и было видно, что пошло не так.
- Все нажатия кнопок в графических клиентах в итоге сводят к выполнению определенных команд консольного клиента, в то же время возможности графических клиентов ограничены по сравнению с консольным

3. Наберите команду ls. Если после этого вы получите список файлов и папок, значит, Bash успешно установилсяи запустился.

```
MINGW64:/c/Users/Admin
 SendTo@
 Untitled.ipynb
 Untitled1.ipynb
 Untitled2.ipynb
 Untitled3.ipynb
 Untitled4.ipynb
 Videos/
'главное меню'@
'Мои документы'@
 Шаблоны@

Admin@File-Server MINGW64 ~
$ ls
 123/                        Cookies@          Links/                 ntuser.dat.L
'3D Objects'/                Desktop/         'Local Settings'@       NTUSER.DAT{d
 Anaconda3/                  Documents/        MicrosoftEdgeBackups/  NTUSER.DAT{d
 AppData/                    Downloads/        Music/                 NTUSER.DAT{d
'Application Data'@          Favorites/        NetHood@               ntuser.ini
'Cisco Packet Tracer 7.0'/   InstallAnywhere/  NTUSER.DAT             OneDrive/
 Contacts/                   lalala/           ntuser.dat.LOG1        OpenVPN/

Admin@File-Server MINGW64 ~
$ |
```

4. Далее для работы нам потребуется создать папку. Создайте её, например, в корне диска d (непосредственно из под Windows), назовите TMP.

Теперь нужно в Bash перейти в эту папку. Для этого используем команду cd (changedirectory):

$ cd /d/tmp/

Нажмите Enter и вы окажетесь в этой папке. Если никаких сообщений об ошибке не выводится, значит, команда выполнена правильно.

```
Admin@File-Server MINGW64 ~
$ cd /d/tmp/

Admin@File-Server MINGW64 /d/tmp
$
```

5. Команда pwd показывает, какая директория текущая в данный момент. Наберите команду и проверьте, где вы находитесь.

```
Admin@File-Server MINGW64 /d/tmp
$ pwd
/d/tmp
```

6. Далее следует задать настройки Git. Они используются для того, чтобы отслеживать авторов изменений. На своем домашнем компьютере следует задать реальные имя, фамилию и email. В учебной лаборатории задайте имя «Пётр ИвановN», «IvanovN@example.com», где N – номер компьютера в лаборатории (используйте свои имя и фамилию).

Обратите внимание на то, что нажимая кнопку ↑ на клавиатуре, можно повторять ранее использованные команды, они будут выводиться в командной строке, после чего их можно редактировать и выполнять. Это значительно ускорит работу.

Попробуйте после ввода имени повторить команду и отредактировать её, задав адрес электронной почты.

```
Admin@File-Server MINGW64 /d/tmp
$ git config --global user.name "Пётр Иванов22"

Admin@File-Server MINGW64 /d/tmp
$ git config --global user.email "Ivanov22@example.com"

Admin@File-Server MINGW64 /d/tmp
$
```

Ключ *–global* означает, что для всех репозиториев будут действовать одни и те же настройки (если задать ключ --local или вообще не задать ключ, настройки будут храниться в данном репозитории и распространяться только на него).

7. Убедитесь, что вы находитесь в папке будущего репозитория (команда *pwd*). Выведите содержимое репозитория (команда *ls*). Убедитесь, что в данный момент папка пуста.

8. Дайте команду

gitinit

Эта команда инициализирует репозиторий в текущей пустой папке, о чем выведется сообщение:

```
Admin@File-Server MINGW64 /d/tmp
$ git init
Initialized empty Git repository in D:/tmp/.git/
```

9. Выполните команду ls. Папка по-прежнему пуста.

Теперь введите ту же команду с ключом –a:

```
Admin@File-Server MINGW64 /d/tmp (master)
$ ls

Admin@File-Server MINGW64 /d/tmp (master)
$ ls -a
./  ../  .git/
```

Вы видите, что в папке появились скрытые папки для служебных целей, созданные Git.

10. Попробуйте на диске D: создать пустую папку GitRepo и перенести туда данную папку tmp(сделайте это непосредственно из папки Мой компьютер, Bash можно не использовать). Далее в GitBashперейдите в эту папку (команда cd).

11. Проверьте, что скрытые файлы по-прежнему на месте, т.е. это по-прежнему репозиторий:

```
Admin@File-Server MINGW64 /d/tmp (master)
$ cd /d/GitRepo/tmp/

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ pwd
/d/GitRepo/tmp

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ ls

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ ls -a
./  ../  .git/
```

12. Дайте команду *git status*.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Эта команда показывает, в каком состоянии в данный момент находится наш репозиторий.

В данном случае Git сообщает, что фиксировать нечего, изменений внутри репозитория не было. Т.о. к абсолютному пути репозиторий не привязан.

Контрольные вопросы:

1. Что такое GitBash?
2. Для чего нужна команда ls?
3. Как сменить директорию?
4. Как отобразить текущую директорию?
5. Для чего задаются настройки Git?
6. Как применить одинаковые настройки для всех репозиториев?

Список литературы:

1. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.
2. Хлебников, А. А. Информационные технологии : учебник / А. А. Хлебников. – М. :КноРус, 2014. – 472 с.

Лабораторная работа №3. Просмотр истории коммитов. Операции отмены.

Цель работы:

Научиться вести историю изменений. Уметь правильно интерпретировать текущий статус репозитория.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Нижний уровень git является так называемой контентно-адресуемой файловой системой. Инструмент командной строки git содержит ряд команд по непосредственной манипуляции этим репозиторием на низком уровне. Эти команды не нужны при нормальной работе с git как с системой контроля версий, но нужны для реализации сложных операций (ремонт повреждённого репозитория и так далее), а также дают возможность создать на базе репозитория git своё приложение.

Практическая часть.

История изменений.

1. Создайте в папке tmp файл README.txt.

2. Вызовите команду git status.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master

No commits yet

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

        README.txt

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

Мы видим, что появился неотслеживаемый файл (Untracked files). Т.е. в нашем репозитории появились посторонние файлы, которые Git видит, но пока не отслеживает.

3. Первое, что нужно сделать при таких изменениях – передать файл под контроль Git. Для этого используется команда git add.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt
```

Будьте внимательны с регистром!

Если команда ничего не вывела, значит, она завершилась успешно.

4. Проверьте git status снова.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master

No commits yet

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

        new file:   README.txt
```

На этот раз Git видит этот файл, и сообщает, что он появился в репозитории. Эти изменения ещё не зафиксированы, но они уже проиндексированы. Т.е. Этот файл готов к тому, чтобы быть зафиксированным.

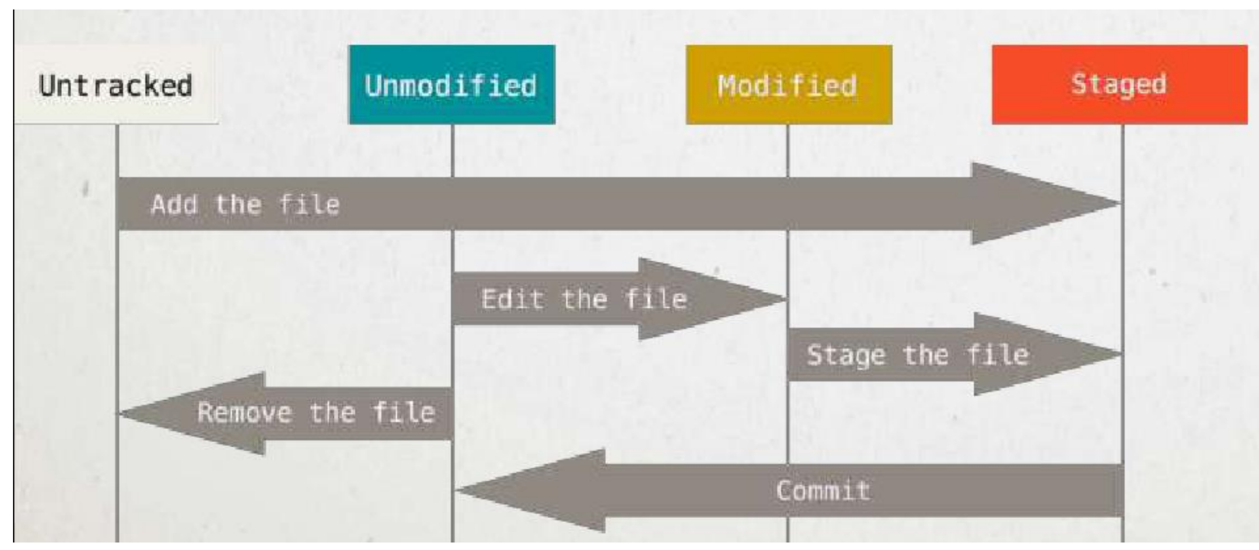
5. Следующий шаг, который мы должны сделать, передав файл под контроль Git и закончив все изменения в нём – закоммитить их (фиксировать изменения).

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git commit -m "Добавлен файл README"
[master (root-commit) 2519868] Добавлен файл README
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.txt
```

Команду gitcommit следует ОБЯЗАТЕЛЬНО использовать с ключом –m и комментарием!

Рассмотрим полученное от Git сообщение. Тут написано, что один файл изменён.

Итак, в Git файлы могут находиться в четырёх состояниях:



1) Untracked – файл просто находится в папке, но для Git он неизвестен, его в истории нет и никогда не было.

2) Staged – подготовленный для фиксации изменений (командой add). Это значит, что файл под контролем Git и он ждёт нашей команды, чтобы зафиксировать изменения (т.е. установить флажок в истории с комментарием).

3) Unmodified–означает, что файл со времени последней фиксации не менялся. Переводится в это состояние фиксацией (commit).

4) Modified – измененный файл. Этот файл уже был в истории, Git о нём знает, файл участвовал в каких-то коммитах, но мы его сейчас изменили. Теперь мы можем либо перевести его в Staged (add), либо отменить изменения.

Наша задача – отчётливо понимать, как происходит перемещение файла по этим состояниям. Результатом работы должно быть состояние файлов Unmodified.

5. Проверьте gitstatus снова. В данный момент все должно быть Unmodified – nothingtocommit.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

6. Откройте файл README.txt и напишите в нем «Hello, world!». Сохраните файл.Проверьте gitstatus снова.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (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:   README.txt

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

Gitсообщает о том, что файл был изменён. Ионнеготовкфиксации (changes not staged for commit).

Каким образом Git это определяет? У него в папке хранятся идеальные копии наших файлов, и он очень быстро их сравнивает с реальными файлами.

7. Готовим изменения к фиксации:

gitaddREADME.txt

8. Проверяем текущий статус.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
```

Видим, что измененный файл *README.txt* готов к фиксации.

9. Далее следует зафиксировать изменения (commit). Не забудьте про ключ и комментарий!!!

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git commit -m "Изменения в README"
[master 49f0cfd] Изменения в README
 1 file changed, 1 insertion(+)
```

Дополнительные команды:

gitrmFILE–удаляет файл из индекса и из рабочей папки;

gitrm --cachedFILE – удаляет файл только из индекса Git.

Оба этих удаления требуют фиксации!

10. Для того, чтобы просмотреть историю коммитов, нужно использовать команду gitlog – это лог всех изменений.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git log
commit 49f0cfdb330fd2d8a4319f16a39e3cb55f0e4b11 (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:34:33 2018 +0300

    Изменения в README

commit 251986888c8f25d0f2c28bc0102b64c0f3cff567
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:00:59 2018 +0300

    Добавлен файл README
```

Здесь мы видим созданные нами два коммита. Длинные числа из шестнадцатеричных цифр – это их номера (кэш). Такие длинные номера нужны для того, чтобы они были уникальными. Они непоследовательны, это сделано специально, для веток (они будут рассмотрены далее).

К каждому коммиту указан автор и дата/время изменений.

11. Создайте в нашей папке ещё один файл test.txt, напишите внутри него TEST. В файле README добавьте пару восклицательных знаков или что-либо ещё. Таким образом мы получили два изменения в репозитории.

12. Проверьте текущий статус.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (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:   README.txt

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

        test.txt

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

Мы видим сообщение о том, что два изменения дожидаются **стейджинга**, т.е. индексации. Для чего вообще выполняется индексация? Таким образом мы показываем, что изменения в данном файле важны и мы планируем их зафиксировать. Если файл не проиндексирован, значит, мы продолжаем его обрабатывать и не уверены, что будем сохранять изменения.

13. Добавляем (индексируем) оба файла:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add test.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
        new file:   test.txt
```

14. Зафиксируем оба изменения в одном коммите:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git commit -m "Тестовый коммит с 2 изменениями"
[master 57f6580] Тестовый коммит с 2 изменениями
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 test.txt
```

Просмотрите историю коммитов в gitlog. Там появился новый только что созданный коммит.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git log
commit 57f658087d95ad4feb553ef3f30309422abba458 (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 12:07:40 2018 +0300

    Тестовый коммит с 2 изменениями

commit 49f0cfdb330fd2d8a4319f16a39e3cb55f0e4b11
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:34:33 2018 +0300

    Изменения в README

commit 251986888c8f25d0f2c28bc0102b64c0f3cff567
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:00:59 2018 +0300

    Добавлен файл README
```

Чтобы более подробно видеть информацию о коммитах, в gitlog можно использовать ключ -p. В этом случае по каждому изменению будет выведено, что было и что стало в каждом измененном файле.

Отмена коммитов.

Крайне нежелательно отменять коммиты, особенно при работе в группе. Это похоже на бухгалтерскую проводку – если документ был проведён, то его нельзя удалить, а можно лишь выполнить коррекцию.

Однако, если очень нужно, то для этого есть специальные команды.

gitresetHEAD– откат к последнему коммиту (здесь HEAD–указатель на текущий коммит). То есть, если что-то пошло не так, то можно отменить все изменения и откатиться к состоянию после последнего коммита.

gitreset --hard<commit> - самая мощная и самая опасная команда – откат к указанному коммиту с потерей всех изменений!

1. Для примера что-нибудь изменим, а затем откатимся к состоянию после коммита. Откройте файл README, замените !!! на ???. Файл test.txt удалите.

2. Выведите текущий статус.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.txt
        deleted:    test.txt

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

Далее выполните gitaddдля обоих файлов, чтобы уж окончательно проиндексировать изменения.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add test.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
        deleted:    test.txt
```

И вдруг в этот самый последний момент мы решаем всё изменить!

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git reset HEAD
Unstaged changes after reset:
M       README.txt
D       test.txt
```

Всё вернулось в состояние до индексации (убедитесь в этом, проверив текущий статус).

3. Далее, чтобы вернуться в состояние до изменений, следует использовать команду gitcheckout--test.txtи gitcheckout--README.txt.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git checkout test.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git checkout README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Убедитесь, что удалённый файл вернулся на место, а во втором отменились изменения.

4. Команда gitreset --hard<commit>, в отличие от предыдущей, работает не по шагам, а разом. Рассмотрим её действие. Удалите и исправьте файлы, как в пункте 1.

Затем примените команду gitreset --hard и проверьте полученный статус:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git reset --hard HEAD
HEAD is now at 57f6580 Тестовый коммит с 2 изменениями

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Всё вернулось на места к состоянию последнего коммита.

5. Попробуем откатиться не на последний, а на предпоследний коммит. Для этого нужно указать его номер (несколько цифр, позволяющих идентифицировать его однозначно). Чтобы знать номера коммитов, сначала выведите лог. Затем выполните откат по номеру предпоследнего коммита:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git log
commit 57f658087d95ad4feb553ef3f30309422abba458 (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 12:07:40 2018 +0300

    Тестовый коммит с 2 изменениями

commit 49f0cfdb330fd2d8a4319f16a39e3cb55f0e4b11
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:34:33 2018 +0300

    Изменения в README

commit 251986888c8f25d0f2c28bc0102b64c0f3cff567
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:00:59 2018 +0300

    Добавлен файл README

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git reset --hard 49f0cfdb
HEAD is now at 49f0cfd Изменения в README
```

Проверьте папку, убедитесь, что всё вернулось к состоянию предпоследнего коммита (второй файл исчез, а в первом только один восклицательный знак).

Примечание:

Если лог не помещается на экране, то программа будет выводить его по частям и после первой части ждать нажатия любой клавиши. Вверх-вниз его можно проматывать стрелками на клавиатуре. Для выхода из этого режима следует нажать «q».

Самостоятельная работа

1. Создайте репозиторий в пустой папке

2. Добавьте в папку файл. Изучите вывод команды git status. Проиндексируйте файл командой git add. Снова посмотрите вывод git status.

3. Зафиксируйте изменения командой git commit.

4. Сделайте и зафиксируйте следующие изменения (каждый подпункт - одна фиксация):

- Добавление сразу трех файлов
- Изменения в тексте двух файлов
- Изменения в тексте одного файла, удаление другого и добавление еще одного

5.* Удалите из какой-либо подпапки все файлы и зафиксируйте это изменение (подпапку, конечно же, надо предварительно создать). А затем удалите и саму пустую подпапку. Объясните результат.

6. Изучите вывод команды git log. Прочтите о ее различных ключах: https://git-scm.com/book/ru/v1/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git-%D0%9F%D1%80%D0%BE%D1%81%D0%BC%D0%BE%D1%82%D1%80-%D0%B8%D1%81%D1%82%D0%BE%D1%80%D0%B8%D0%B8-%D0%BA%D0%BE%D0%BC%D0%BC%D0%B8%D1%82%D0%BE%D0%B2

7. * Создайте свой формат вывода git log с помощью ключа --pretty=format

8. * Внесите изменения в репозиторий, но не фиксируйте их. Посмотрите git status. Теперь дайте команду git stash и снова посмотрите статус и свои файлы. А теперь git stash apply. Попробуйте объяснить результат, не прибегая к документации.

В отчёт приложите все использованные вами команды и их вывод в консоль.

В отчёте опишите, как вы поняли принцип действия всех новых команд из самостоятельной работы.

Задания, помеченные знаком "*" требуют самостоятельного поиска дополнительных материалов.

Контрольные вопросы:

1. Как отобразить текущий статус репозитория?
2. Какие файлы называются Untracked files?
3. Как передать файл под контроль Git?
4. Как отобразить список изменений в репозитории?
5. Как отменить коммит и почему это делать не рекомендуется?
6. Как Git идентифицирует все этапы изменения?

Список литературы:

1. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.
2. Хлебников, А. А. Информационные технологии : учебник / А. А. Хлебников. – М. :КноРус, 2014. – 472 с.

Лабораторная работа №4. Работа с удалёнными репозиториями

Цель работы:

Научиться работать с удаленными репозиториями. Загрузка веток из других источников. Операции с удаленными ветками

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Удалённые ветки — это ссылки на состояние веток в ваших удалённых репозиториях. Это локальные ветки, которые нельзя перемещать; они двигаются автоматически всякий раз, когда вы осуществляете связь по сети. Удалённые ветки действуют как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

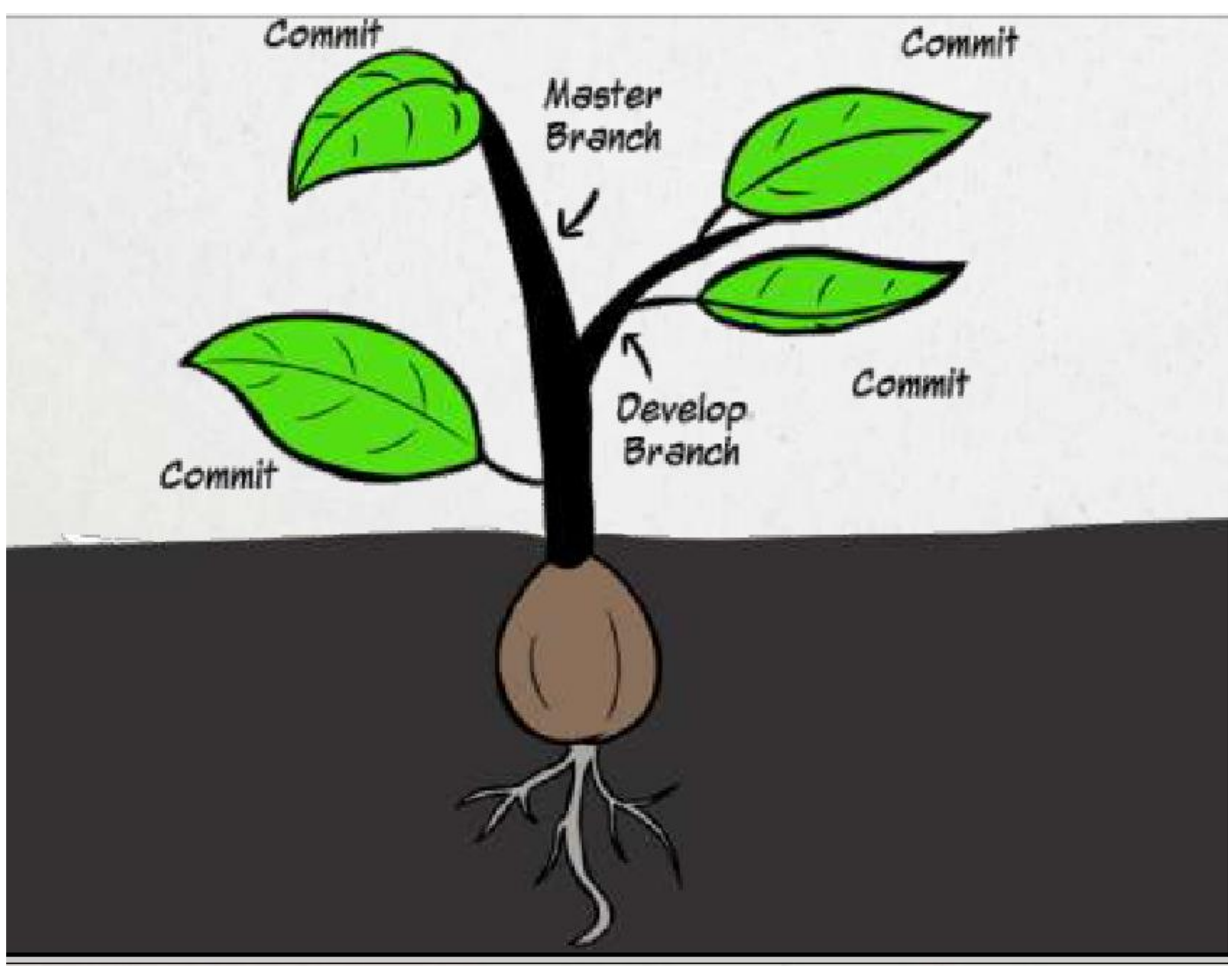
Они выглядят как (имя удал. репоз.)/(ветка). Например, если вы хотите посмотреть, как выглядела ветка master на сервере origin во время последнего соединения с ним, проверьте ветку origin/master. Если вы с партнёром работали над одной проблемой, и он выложил веткуiss53, у вас может быть своя локальная ветка iss53; но та ветка на сервере будет указывать на коммит в origin/iss53.

Практическая часть.

Когда на экран выводится git log, мы видим слово (master). Это – основная ветвь истории изменений. На данный момент она для нас единственная.

Ветка – это последовательность коммитов.

1. Дайте команду git status в своём репозитории. Будет выведено сообщение «On branch master» - «Я нахожусь на ветке master» - в главной ветви, которая начинается с самого первого коммита в истории и доходит до последнего.



В данной работе мы будем рассматривать только ветвь master.

Git – это распределенная система, это значит, что у нас может быть много репозиториев. Они могут быть распределены – находиться на разных компьютерах внутри локальной сети, в разных местах одного компьютера или где-то далеко в интернете. Распределенность Git заключается в том, что он умеет определенным образом связывать эти репозитории между собой.

В некоторых СКВ система репозиториев клиент-серверная, но не в Git. Здесь все репозитории равноправны между собой.

Репозиотрии делятся ан 2 типа – локальные (у нас на компьютере) и удалённые (где-то ещё). Git умеет связывать репозитории между собой. Цель этой лабораторной работы – изучить данную технологию, то, как эта связь выполняется.

Для работы с удаленными репозиториями используется команда git remote.

2. Создадим в папке GitRepo новый чистый репозиторий tmp2, а старый удалим. В Git Bash перейдем в новую папку (cd <путь>).

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ cd ..

Admin@File-Server MINGW64 /d/GitRepo
$ cd /d/GitRepo/tmp2

Admin@File-Server MINGW64 /d/GitRepo/tmp2
$ |
```

Инициализируйте репозиторий (git init).

3. Создайте аналогично второй репозиторий в корне диска D:

```
MINGW64:/d/Repo2
$ cd ..

Admin@File-Server MINGW64 /d/GitRepo
$ cd /d/GitRepo/tmp2

Admin@File-Server MINGW64 /d/GitRepo/tmp2
$ git init
Initialized empty Git repository in D:/GitRepo/tmp2/.git/

Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ cd /d/Repo2/

Admin@File-Server MINGW64 /d/Repo2
$ git init
Initialized empty Git repository in D:/Repo2/.git/

Admin@File-Server MINGW64 /d/Repo2 (master)
$ |
```

Добавьте в него файл README1.txt с каким-либо текстом, сделайте коммит. Затем в Git Bash сделайте текущим первый репозиторий.

```
Admin@File-Server MINGW64 /d/Repo2 (master)
$ git add README1.txt

Admin@File-Server MINGW64 /d/Repo2 (master)
$ git commit -m "Новый файл в Repo2"
[master (root-commit) 928ae36] Новый файл в Repo2
 1 file changed, 1 insertion(+)
 create mode 100644 README1.txt

Admin@File-Server MINGW64 /d/Repo2 (master)
$ cd /d/GitRepo/tmp2/
```

4. Команда *git remote add* добавляет к текущему репозиторию ссылку на удалённый (находящийся в другом месте) под каким-то именем, назовём его super2.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git remote add super2 "d:\Repo2"
```

5. Команда git remote покажет нам список прикрепленных к нашему репозиторию удалённых репозиториев.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git remote
super2
```

С ключом –v вывод будет немного подробнее:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git remote -v
super2  d:\Repo2 (fetch)
super2  d:\Repo2 (push)
```

Показывается не только название, но ещё и адрес удалённого репозитория.

Связь создана, однако на статус это никак не повлияет.

Для просмотра удаленного репозитория можно использовать команду

git remote show super2

6. Далее выполним команду git fetch super2

Эта команда возьмет информацию обо всех изменениях в удалённом репозитории и перенесёт в текущий.

Обратите внимание, что <u>будет перенесена только информация об изменениях</u>, а не они сами!!! fetch не меняет файлы!!!

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git fetch super2
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From d:\Repo2
 * [new branch]      master     -> super2/master
```

Выведите содержимое репозитория командой ls. Вы увидите, что он не изменился. В нем по-прежнему ничего нет. Однако, теперь мы знаем всё об удалённом репозитории.

После того, как мы сделали git fetch, мы получили в том числе список веток удалённого репозитория. В данный момент у нас есть одна ветка – master. А в удаленном репозитории может быть много разных веток. Для того, чтобы работать с удаленными изменениями, нам нужно связать ветки между собой. В самом простом случае – связать две ветки master между собой.

7. Выполните команду

git checkout --track super2/master

Здесь сказано, что Git должен создать связь текущей ветки данного репозитория с ветвью master репозитория super2.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git checkout --track super2/master
Already on 'master'
Branch 'master' set up to track remote branch 'master' from 'super2'.
```

В ответ Git пишет, что ветка master установлена для отслеживания удалённой ветки master из super2.

8. Команда git pull – команда, которая получает изменения.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git pull
Already up to date.
```

Т.е. Эта команда позволяет выкачать все изменения из удаленного репозитория.

В следующей работе рассмотрим использование для этих целей популярного в настоящее время сервиса Github.

Контрольные вопросы:

1. Что такое ветка?
2. На какие типы делятся репозитории Git?
3. Как добавить ссылку на удаленный репозиторий?
4. Как вывести подробный список прикрепленных репозиториев?
5. Что выполняет команда *git remote show*?

Список литературы:

1. Хлебников, А. А. Информационные технологии : учебник / А. А. Хлебников. – М. :КноРус, 2014. – 472 с.
2. Золотов, С.Ю. Проектирование информационных систем : учебное пособие / С.Ю. Золотов ; Министерство образования и науки Российской Федерации, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). - Томск : Эль Контент, 2013. - 88 с. : табл., схем. - ISBN 978-5-4332-0083-8 ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=208706

Лабораторная работа №5. Работа с метками.

Цель работы:

Научиться присваивать метки к репозиториям. Работа с сабмодулями

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Как и большинство СКВ, Git имеет возможность помечать (tag) определённые моменты в истории как важные. Как правило, этот функционал используется для отметки моментов выпуска версий (v1.0, и т.п.). В этом разделе вы узнаете, как посмотреть имеющиеся метки (tag), как создать новые. А также вы узнаете, что из себя представляют разные типы меток.

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

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

Практическая часть.

Теги

Теги – это метки, которыми можно пометить коммит. Они часто применяются в работе.

Есть лёгкие теги. Они очень простые. Мы берем команду gittagи придумываем какую-либо метку. Часто теги используют, чтобы помечать какие-либо версии. Например, показать, что на каком-то коммите мы

достигли версии 1.0. То есть, это просто ещё одно имя для коммита, для того, чтобы пометить какие-то важные в истории коммиты.

1. Введите команду gittag– это команда, которая выводит состояние меток в текущем репозитории. В данном случае меток нет, ничего не будет выведено.

2. Вызовите команду gitstatus, убедитесь, что вы находитесь в ветке master. Теперь пометим тегом 1.0 текущий коммит.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master
nothing to commit, working tree clean

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git tag 1.0
```

3. Повторите команду gittag. Вы увидите, что у вас появился тег.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git tag
1.0
```

4. Если вы хотите увидеть помеченный коммит, используйте команду gitshow.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git show 1.0
commit ecd8cd72b1ebaf1946bc89ac90c50223d3065971 (HEAD -> ma
re/1)
Merge: fe36a38 2374d7c
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Oct 2 14:31:58 2018 +0300

    Конфликт решен

diff --cc first.txt
index 5ace0b5,cdc9555..51bd26d
--- a/first.txt
+++ b/first.txt
@@@ -1,3 -1,3 +1,3 @@@
  Это первая строчка.

- Красная кнопка
 -Тут две зелёных кнопки.
++Одна красная и две зеленые кнопки.
```

Вы видите, что в этом коммите был решен конфликт – проблема с красной и зелеными кнопками.

5. Меток в проекте может быть множество, поэтому у команды git tag может быть метка -l – маска. Например:

git tag -l ‘1.*’

Эта маска покажет, какие метки начинаются с «1.»

6. Для удаления тега применяется команда gittag -d.Например:

git tag -d 1.0

Кроме лёгких тегов существуют **аннотированные метки**. Это тоже метка (имя для коммита), однако, кроме имени в ней можно поместить

некоторое сообщение, дату и время создания, имя того, кто создал, и даже подписать цифровой подписью.

Это достаточно сложная возможность, она используется в больших проектах, где много разработчиков сливают свои изменения, и есть один главный разработчик, которому поручено принимать решения о выпуске продукта. Они используются для того, чтобы пометить коммит готового к сборке продукта. Этот человек (релиз-мастер) с помощью своей цифровой подписи свидетельствует о том, что он даёт визу на публикацию. Он ставит метку на этот коммит и подписывает её. Автоматические системы сборки на это реагируют и начинают сборку продукта (эти технологии не слишком используются обычными разработчиками).

Выглядит эта команда так:

git teg -a 1.0 –m “Тест тега”

7. Создайте такой тег.
8. Используйте команду git tag 1.0. В списке тегов вы увидите этот тег в обычном виде.
9. Вызовите команду git show 1.0. Вы увидите намного больше информации об этом коммите, чем с обычным тегом, в том числе (сразу после команды) сведения о том, кто эту метку поставил (строчка Tagger), когда он её поставил (строчка Date). Если бы мы воспользовались при создании тега своим ключом, то тут бы была также информация о нём, и некоторый автоматизированный софт смог бы проверить её подлинность. Это используется в очень крупных проектах, например, при работе над Линуксом.

Метки можно создавать и позже. Достаточно при создании указать номер уже существующего коммита:

git tag -a 1.0 -m "Сообщение" 1baf

10. Выведите список коммитов.
11. Создайте тег 1.1 для созданного ранее коммита.

Метки по умолчанию не передаются в удаленный репозиторий. Для передачи используйте команды:

git push origin 1.0 – ДЛЯ КОНКРЕТНОЙ МЕТКИ

git push origin --tags - ДЛЯ ВСЕХ ТЕГОВ

Это происходит, так как git push отправляет изменения, а метка изменением не является.

СУБМОДУЛИ

СУБМОДУЛЬ – это, фактически, репозиторий внутри вашего репозитория.

git submodule add РЕПОЗИТОРИЙ ПАПКА

Для чего он может быть нужен? Например, вы работаете над проектом и нашли какую-то библиотеку, которую хотите к нему подключить (например, нашли её на github). Однако, её нельзя просто склонировать на ваш компьютер, перетащить файлы к себе и закоммитить. Однако, в этом случае, если автор внесёт какие-то коррективы в свою библиотеку, вы об этом никогда не узнаете.

Хорошим подходом в этом случае является субмодуль. Т.е. Можно просто включить в ваш репозиторий ссылку на другой.

1. Создайте на hithub под своим профилем второй репозиторий, добавьте в него файлы и включите его в ваш текущий проект. Для этого после того, как скопировали ссылку на репозиторий, перейдите в Git Bash в текущий репозиторий и выполните команду

git submodule add ____вставить адрес___ app

(app – название нового репозитория)

2. Откройте папку с репозиторием и убедитесь, что там появилась новая папка с субмодулем.

3. Проверьте git status. Вы видите, что добавился автоматически файл gitmodules, который описывает все подключенный модули, его можно прочитать (cat .gitmodules). Также добавилась папка app, которая в статусе отображается как одно целое, хотя внутри она содержит другие файлы и папки библиотеки.

4. Сделайте коммит и у вас зафиксируется информация о том, что в папке app находится клон удалённого репозитория.

После добавления субмодуля нужно зафиксировать изменения, обычной командой git commit

5. Сделайте коммит.

Есть ещё две необходимые команды для работы с субмодулями:

git submodule init Инициализирует субмодули в репозитории, где это еще не сделано (например сразу после клонирования)

git submodule update

• Загружает файлы субмодулей

Контрольные вопросы:

1. Что такое метка?
2. Какие типы меток существует?
3. Как добавляется метка к репозиторию?
4. Как вывести информацию по метке?

5. Для чего нужен сабмодуль?
6. Как добавить сабмодуль?

Список литературы:

1. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.
2. Хлебников, А. А. Информационные технологии : учебник / А. А. Хлебников. – М. :КноРус, 2014. – 472 с.

Лабораторная работа №6. Работа с ветками

Цель работы:

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Команда git branch делает несколько больше, чем просто создаёт и удаляет ветки. При запуске без параметров, вы получите простой список имеющихся у вас веток:

```
$ git branch
  iss53
* master
  testing
```

Обратите внимание на символ *, стоящий перед веткой master: он указывает на ветку, на которой вы находитесь в настоящий момент (т.е. ветку, на которую указывает HEAD). Это означает, что если вы сейчас выполните коммит, ветка master переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду git branch -v:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Практическая часть.

Понятие «Ветки».

Репозиторий хранит в себе информацию о состояниях репозитория. Они называются «снапшоты». Каждое состояние – это коммит. Коммит – это ***снапшот и данные об авторстве, времени и номере коммита, а также указатель на родительский коммит***.Снапшот – это информация о том,

каким был репозиторий в какой-то момент времени. Это полная информация обо всех файлах и папках, снимок файловой системы. Он тщательно упакован и занимает минимально возможное место.

В отличие от других систем контроля версий, это не запись об изменениях от предыдущего состояния, а вся информация о репозитории. Это позволяет быстро перемещаться по коммитам, не внося изменения в несколько этапов, а просто перейдя на нужный коммит.

Это также позволяет в случае какой-то потери данных пропустить битый коммит и перейти к предыдущей рабочей версии.

Коммит снабжен указателями на его родительские коммиты:

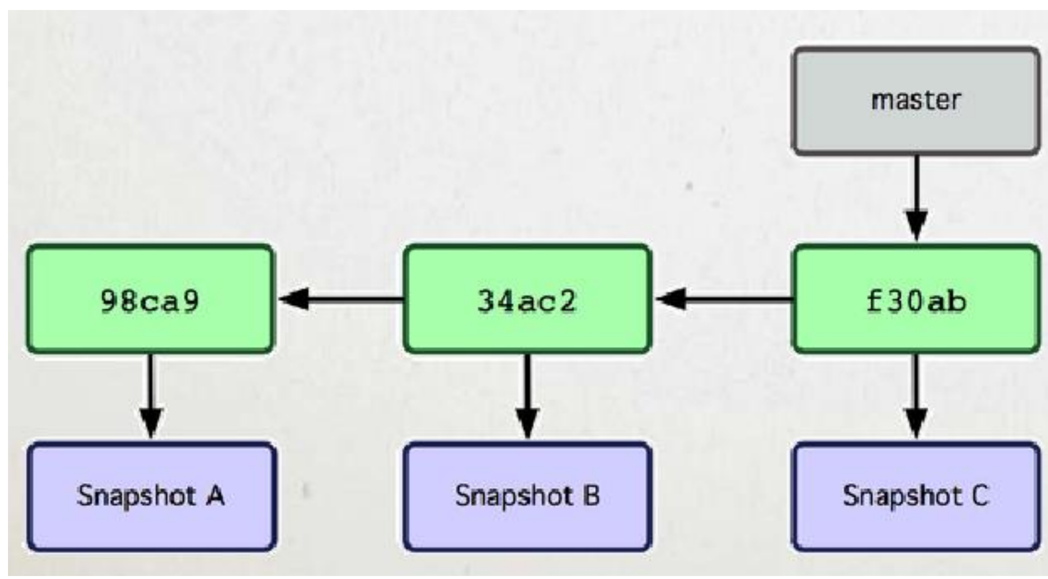
- Ноль указателей – если это первый коммит;
- Один – если это обычный;
- Несколько – в случае слияния изменений (об этом пойдет речь позже)

Таким образом, система всех этих указателей на родительские коммиты создает историю проекта, где в самом конце стоит самый первый коммит.

Ветка – это указатель на какой-либо коммит в истории.

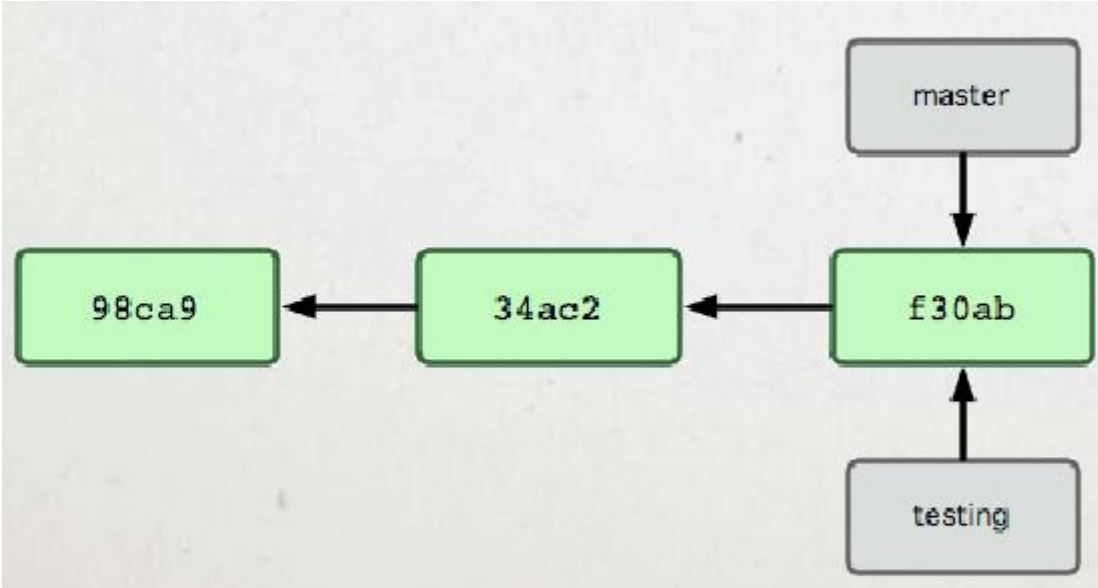
При создании репозитория автоматически создается ветка master. Указатель на нее автоматически сдвигается при каждом коммите. Ветка master считается основной, хотя это просто договорённость.

На рисунке представлено перемещение указателя «master». Сначала был создан снапшот А и указатель мастер был на нем, потом создали снапшотB, указатель master переместился на него, затем перешел на станпшот С:



Репозитории в Git могут содержать неограниченное число новых веток. Для того, чтобы создать новую ветку, нужно дать команду gitbrunch<новая ветка>.

Например, команда gitbrunchtesting создаст новую ветку testing, то есть создаст новый указатель на текущее состояние репозитория и даст ему имя testing. Больше эта команда ничего не делает! В результате история будет выглядеть следующим образом:



Таким образом, на один и тот же коммит могут указывать сколько угодно веток.

Перейдем к практике.

1. Создайте на диске dпапку ProGit и в ней папку brunches:

```
Танюшка@NoteBook MINGW64 ~
 cd /d/

Танюшка@NoteBook MINGW64 /d
 mkdir ProGit

Танюшка@NoteBook MINGW64 /d
 mkdir /d/ProGit/branches/

Танюшка@NoteBook MINGW64 /d
 cd /d/ProGit/branches/

Танюшка@NoteBook MINGW64 /d/ProGit/branches
```

2. Инициализируйте репозиторий в этой папке

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git init
Initialized empty Git repository in D:/ProGit/branches/.git/
```

3. Проверьте статус репозитория, убедитесь, что он совершенно пуст. Мы находимся на ветке master (Onbranchmaster).

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

4. Создайте из-под Windows в этой папке какой-либо текстовый файл (например, README.txt), напишите в него текст «Проверка веток».

5. Добавьте этот файл и сохраните коммит под названием «Первый коммит».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git add ./README.txt

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git commit -m "Первый коммит"
[master (root-commit) 05b133a] Первый коммит
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```

6. Проверьте статус репозитория:

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master
nothing to commit, working tree clean
```

Мы по-прежнему находимся в ветви master.

7. Вызовите git log, и вы увидите, что существует только один «Первый коммит».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

8. Теперь создаем новую ветку.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git branch testing
```

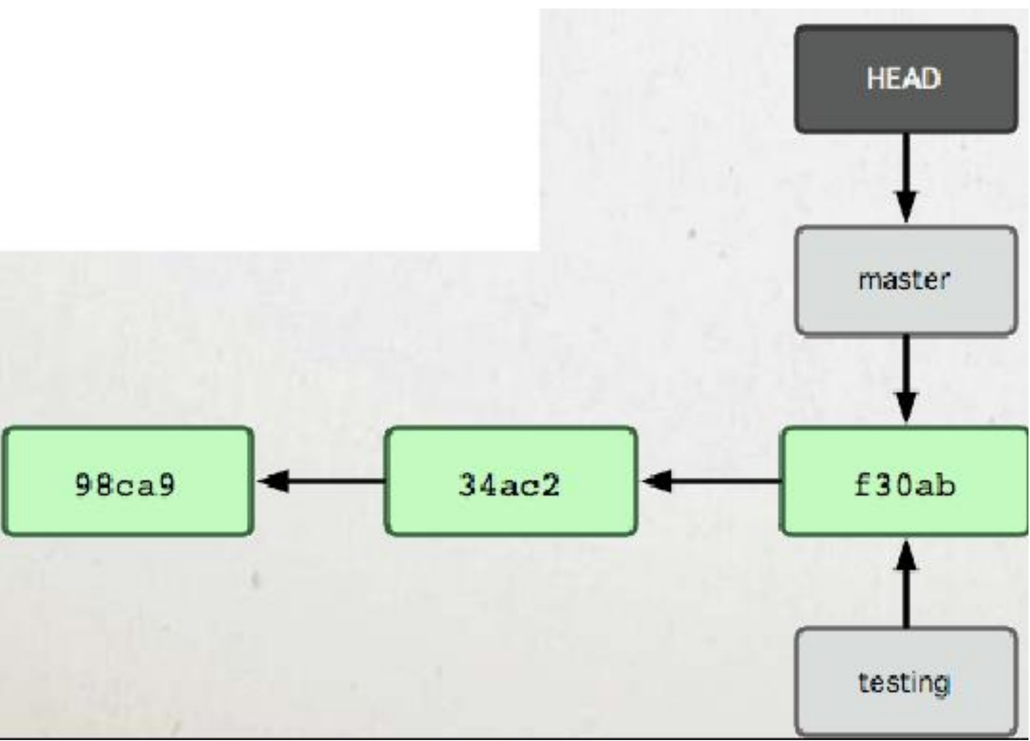
Мы создали новую ветку, которая указывает на тот же коммит, который на скриншоте виден под номером 05b133ad74…

9. Однако, мы до сих пор находимся в ветке master!!!Проверьте это командой git status.Это потому, что команда git branch ветку только создает.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master
nothing to commit, working tree clean
```

10. Для того, чтобы переключиться на новую ветку, используется команда git checkout. Для её понимания рассмотрим, что такое HEAD.

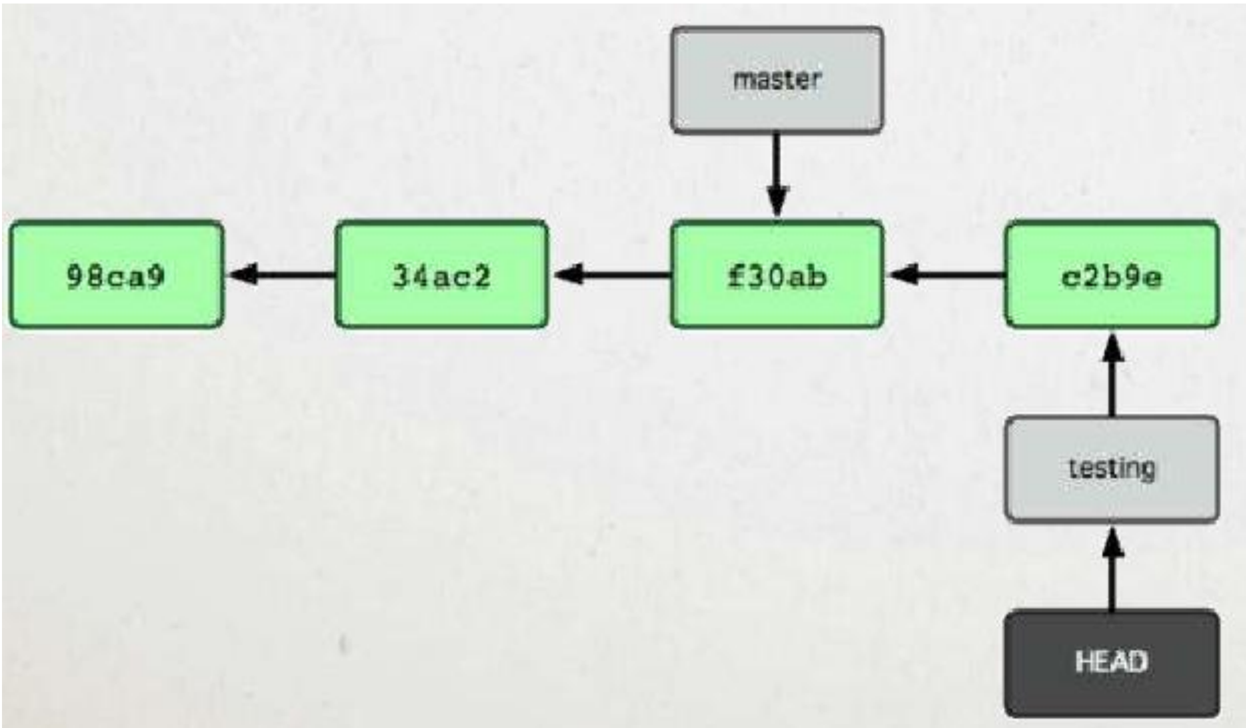
HEAD – это указатель на указатель. Это специальный указатель на текущую ветку и коммит. То есть, это указатель на текущее состояние репозитория (говорят, что это указатель на то, в какой ветке мы находимся. На самом деле мы в ней не находимся, просто текущий указатель указывает на нее). Команда git checkout передвинет HEADна указанную нами ветку.



11. Перейдите на ветку testing.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout testing
Switched to branch 'testing'
```

В результате предыдущая схема будет выглядеть так:



12. В консоли вы видите текст: «Switched to branch ‘testing’», это означает, что мы переключились на ветку testing.

13. Также можно проверить текущую ветку с помощью gitstatus:

```
Танюшка@NoteBook MINGW64 /d/temp/branches
 git status
On branch testing
nothing to commit, working tree clean
```

Как видите, никаких изменений не произошло, мы просто переключились с одной ветки на другую.

14. Сделаем в ветке testing ещё один коммит. Для этого откройте файл README.txt и допишите в конце любое слово, например, «Тест». Добавьте и индексируйте изменения, сохранив коммит под именем «Коммит в ветке testing».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git add ./README.txt

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git commit -m "Коммит в ветке testing"
[testing 450680d] Коммит в ветке testing
 1 file changed, 2 insertions(+), 1 deletion(-)
```

15. Проверьте git status, убедитесь, что все изменения зафиксированы. Выведите git log и вы увидите два коммита.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 450680d12f80565cea4c641d8c49136e7ce3ddd5 (HEAD -> testing)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:39:30 2018 +0300

    Коммит в ветке testing

commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

16. Далее перейдите в ветку master.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout master
Switched to branch 'master'
```

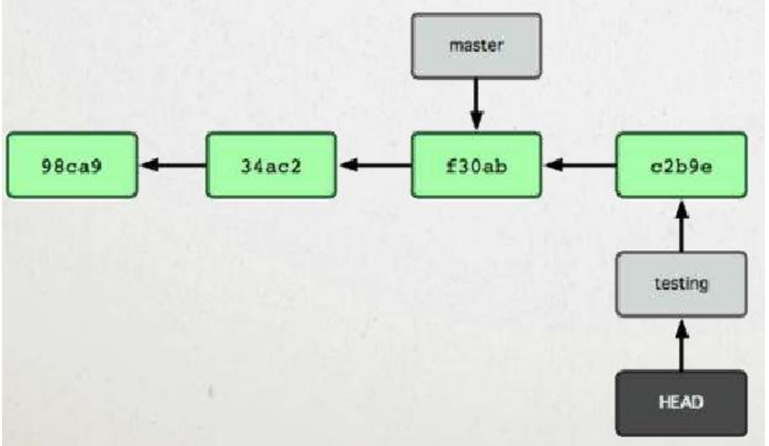
Мы вернули репозиторий в то состояние, с которым была связана ветка master. Чтобы убедиться в этом, откройте файл README.txt, и вы увидите, что дописанный текст «Тест», который был сохранен в ветке testing, исчез. Он существует только в ветке testing.

17. Выведите git log. Вы увидите, что коммит, относящийся к ветке testing, также не отображается.

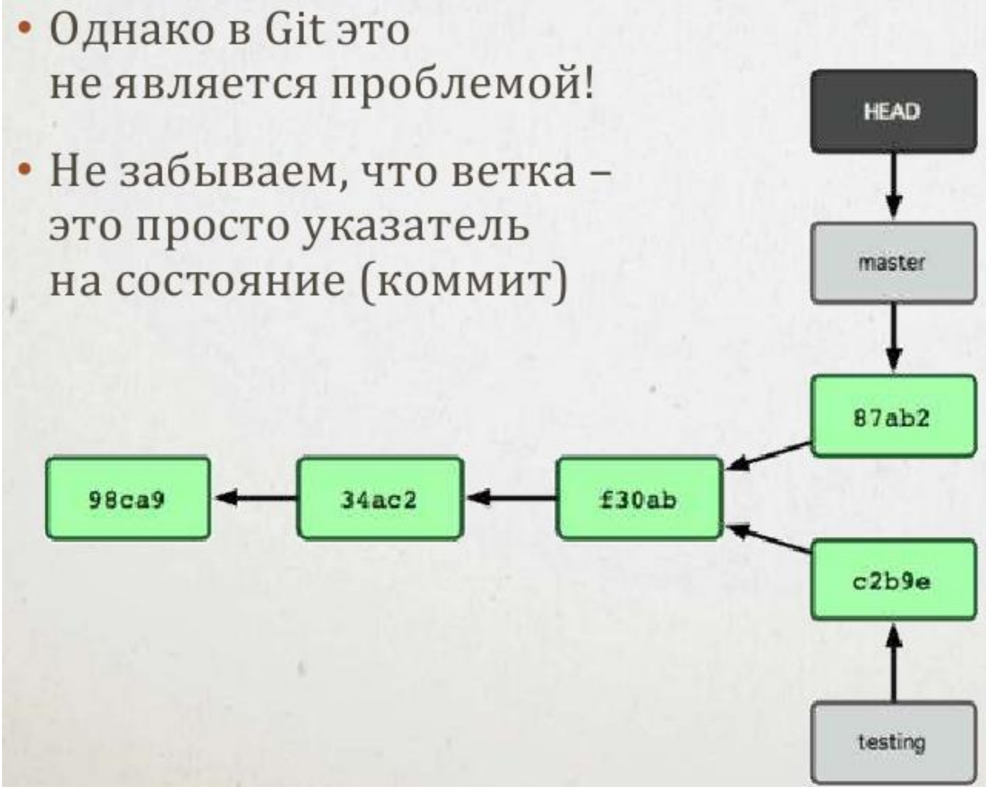
```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

18. Перейдите опять на ветку testing. Проверьте лог, вы снова увидите два коммита. Откройте файл README.txt, там опять будет две строчки.

Итак, мы находимся в ситуации, представленной на рисунке:



Возникает вопрос, что произойдет, если мы перейдем на ветку master и создадим ещё один коммит? Получится, что у одного из коммитов несколько ветвей:



19. Промоделируем эту ситуацию в нашем репозитории. Перейдите на ветку master и выведите лог.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout master
Switched to branch 'master'

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

20. Внесем изменения в репозиторий. Добавьте файл index.txt, напишите в него текст «Главный файл».

21. Создайте коммит «Коммит в ветке master».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git add ./index.txt

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git commit -m "Коммит в ветке master"
[master e915b88] Коммит в ветке master
 1 file changed, 1 insertion(+)
 create mode 100644 index.txt
```

22. Вызовите git log. Вы увидите в ветке master два коммита:

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit e915b88e7832956d5713b8c762b1e3cdc7c51929 (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Fri Sep 21 12:20:54 2018 +0300

    Коммит в ветке master

commit 05b133ad74ed028634f770e1dd94067aae6d9cbb
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

23. Переключитесь в ветку testing и также посмотрите git log. Вы видите, что второй коммит совершенно другой:

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout testing
Switched to branch 'testing'

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 450680d12f80565cea4c641d8c49136e7ce3ddd5 (HEAD -> testing)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:39:30 2018 +0300

    Коммит в ветке testing

commit 05b133ad74ed028634f770e1dd94067aae6d9cbb
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

Таким образом мы получили разветвление в истории.

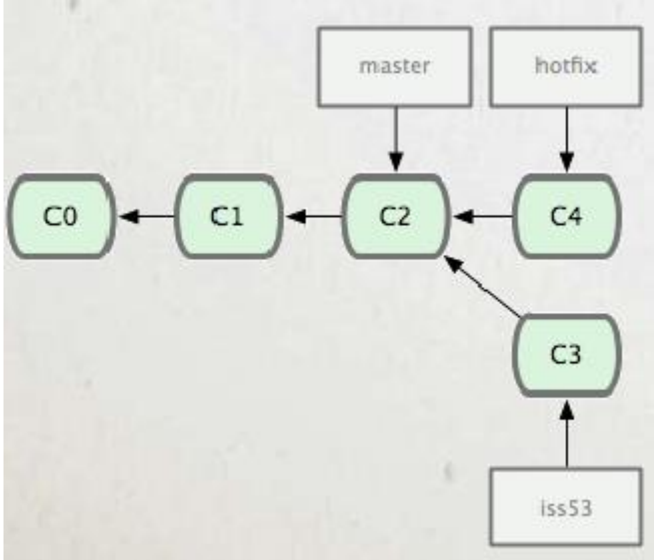
Важная рекомендация: переключаться между ветками следует в чистом состоянии репозитория! Не оставляйте непроиндексированных изменений! Иначе могут возникнуть проблемы: не произойдет переключения, пропадут незафиксированные данные и др.

Однако, если изменения пока не подлежат коммиту, стоит использовать git stash, переключиться, а по возвращении использовать git stash apply. Это стандартный метод работы с Git!!!

Слияние веток

Допустим, мы хотим объединить две ветки, слив изменения в одну.

Например, у нас в репозитории ситуация, представленная на рисунке:



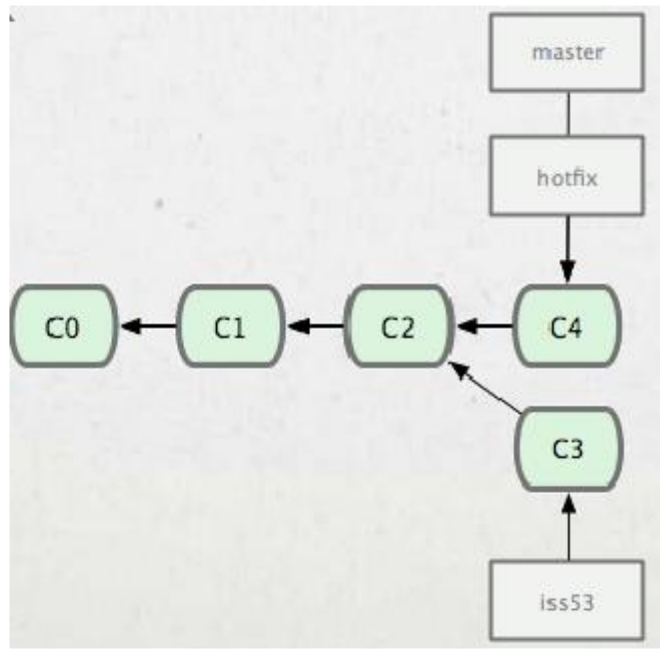
Здесь две ветки – master и hotfix находятся на одной линии истории (iss53 в данный момент не рассматриваем). То есть, в какой-то момент мы сделали ветку hotfix, внесли в нее какие-то изменения, закоммитили их, протестировали, и теперь готовы эти изменения влить в master.

Для этого нужно:

1) переключиться на ветку master (на ту ветку, которая будет принимать изменения);

2) дать команду git merge hotfix (merge – означает «слить»).

После этого Git рассуждает следующим образом: master и hotfix находятся на одной ветке истории. Для слияния их изменений достаточно историю просто перемотать вперёд. Что он и сделает. То есть, в данном случае никакого слияния фактически происходить не будет. Просто указатель мастер тоже переместится на C4:



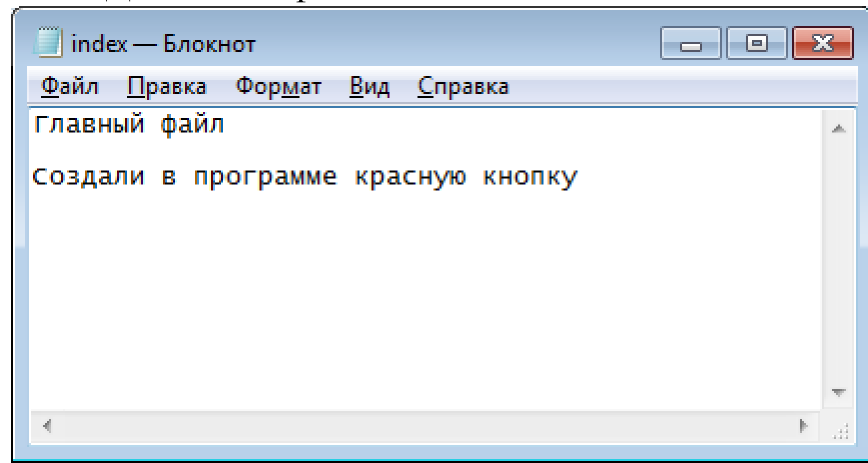
Такой тип слияния называется fast-forward (ff). Такой сценарий реализуется только тогда, когда у нас ветки находятся на одной линии истории.

После этого ненужную ветку можно сразу удалить. Для этого используется команда git branch -d *hotfix* (вместо hotfix подставить название своей удаляемой ветки).

Рассмотрим пример.

1. Выведите в ветке master список коммитов, вы увидите 2 коммита.
2. Создадим от этой ветки ветку hotfix (например, нам надо срочно внести какое-то изменение в программу).

3. Переключитесь на эту ветку.
4. Добавьте в файле index новый текст:



5. Проверьте git status, вы увидите, что появились изменения.
6. Сделайте коммит «Создали красную кнопку».
7. Проверьте лог, вы увидите все три коммита. При этом третий коммит относится только к ветке hotfix:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (hotfix)
$ git log
commit c7863fb49abecb7270696496008e8c0d00eae877 (HEAD -> hotfix)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:53:16 2018 +0300

    Создали красную кнопку

commit b2052d1d9887cb4ebacf27c92f56f54603bb9d47 (master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:44:23 2018 +0300

    Коммит в ветке master

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

Допустим, мы внесли таким образом нужные изменения в нашу программу, их в ветке hotfix изучили инженеры по качеству, тестировщики и

так далее, после чего, когда эта версия принята, вам как программисту поступает команда влить эти изменения в основную ветку.

8. Переключитесь на ветку master и выведите лог. Вы увидите два коммита ветки master.

9. Выполните слияние:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git merge hotfix
Updating b2052d1..c7863fb
Fast-forward
 index.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Вы видите, что в 4й строке написано Fast-forward. Это означает, что в результате команды произошла перемотка этой истории, и в её ходе был изменен файл index.txt (5я строка).

10. Проверьте статус. Мы в ветке master, коммиты не требуются.

11. Выведите лог. Вы увидите, что все коммиты появились в ветке master.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git log
commit c7863fb49abecb7270696496008e8c0d00eae877 (HEAD -> master, hotfix)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:53:16 2018 +0300

    Создали красную кнопку

commit b2052d1d9887cb4ebacf27c92f56f54603bb9d47
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:44:23 2018 +0300

    Коммит в ветке master

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

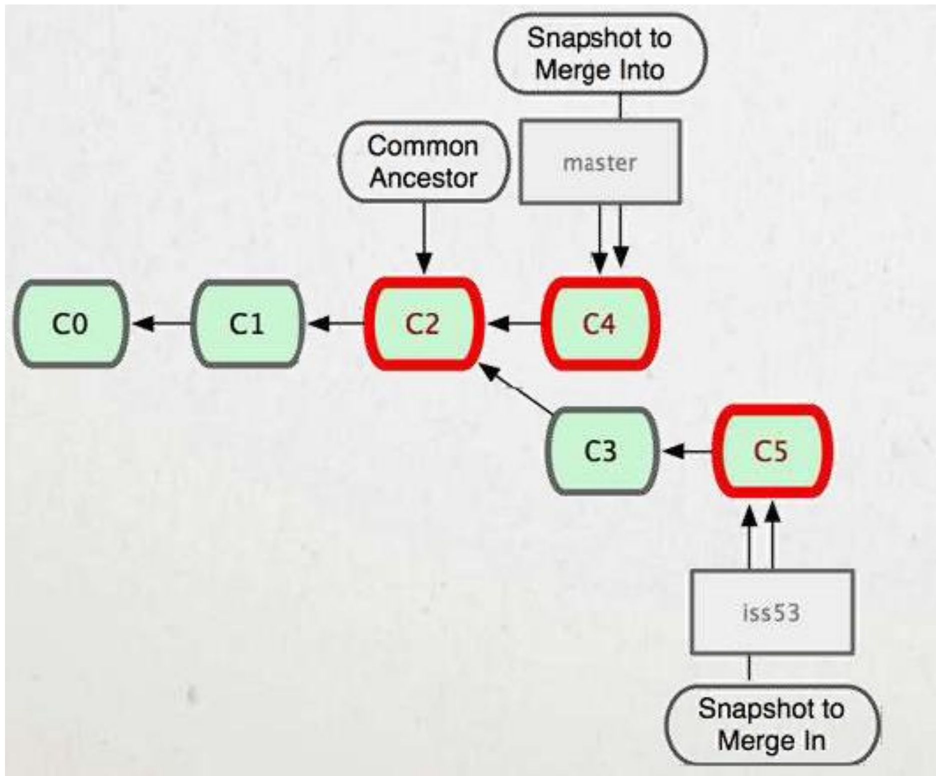
Таким образом, мы рассмотрели самый простой вид слияния – Fast forward.

12. Вызовите команду git branch –v

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git branch -v
  hotfix  c7863fb Создали красную кнопку
* master  c7863fb Создали красную кнопку
  testing 4d6079e Коммит в ветке testing
```

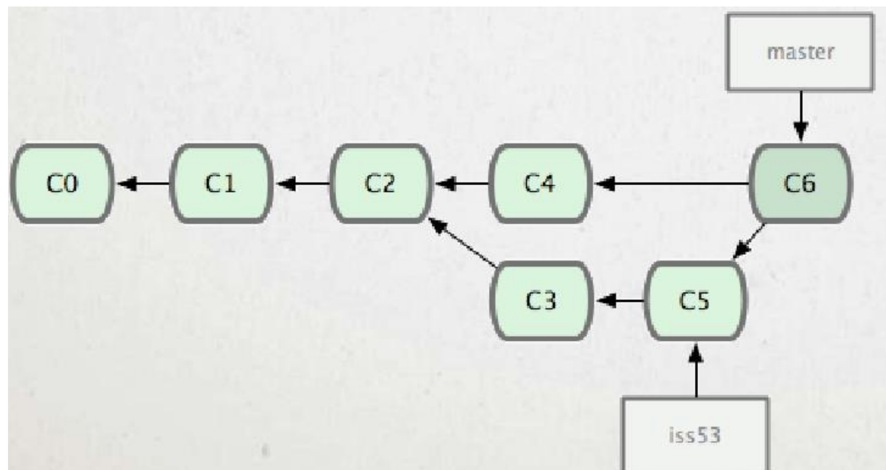
Этак команда выводит полный перечень веток репозитория. Звездочкой отмечена текущая ветка, на которую смотрит HEAD.

Теперь рассмотрим более сложный случай: как сливать изменения, если ветки уже разошлись? Допустим, что ветка master уже ушла вперёд от общего родителя (C4). А вторая ветка Iss53 уже ушла от неё на два коммита (C5). То есть, эти ветки не находятся на одной линии истории.



В этом случае git применяет алгоритм «Наилучшего общего предка». От обеих веток git просматривает историю назад, пока не найдет ближайшего общего предка. После этого он начинает последовательно к этому предку применять коммиты, ориентируясь на время, когда они были созданы, и таким образом создает новый коммит слияния.

Таким образом, в отличие от Fast forward, когда ничего нового не создавалось, здесь создастся новое изменение. Создастся новый коммит C6 в ветке master, который соберет в себе все изменения C3, C4 и C5:



Рассмотрим практический пример.

13. Проверьте статус, вы должны находиться в ветке master:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git status
On branch master
nothing to commit, working tree clean
```

14. Она уже ушла достаточно далеко от ветки testing, созданной ранее. Убедимся в этом, выведем лог для ветки master, затем перейдем в ветку testing и проверим лог для неё.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git log
commit c7863fb49abecb7270696496008e8c0d00eae877 (HEAD -> master, hotfix)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:53:16 2018 +0300

    Создали красную кнопку

commit b2052d1d9887cb4ebacf27c92f56f54603bb9d47
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:44:23 2018 +0300

    Коммит в ветке master

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

Переходим в testing:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git checkout testing
Switched to branch 'testing'

d103@S-T000012680 MINGW64 /d/ProGit/branches (testing)
$ git log
commit 4d6079ec915327ceb8f4f719587e0d0ffede2ebe (HEAD -> testing)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:41:53 2018 +0300

    Коммит в ветке testing

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

У них общий только первый коммит, в остальном они разошлись, у каждого своя история.

15. Снова переключаемся в master и вливаем неё ветку testing (git merge testing). Начнется слияние:

```
MINGW64:/d/ProGit/branches
Merge branch 'testing'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
D:/ProGit/branches/.git/MERGE_MSG [unix] (10:21 22/09/2018)          1,1 Весь
"D:/ProGit/branches/.git/MERGE_MSG" [unix] 7L, 250C
```

16. Слияние удалось и нам открылся текстовый редактор, который просит нас ввести сообщение для коммита. Чтобы выйти из него, нажмите последовательно <esc> <:> <q> (если в качестве текстового редактора по умолчанию у вас установлен другой, то вы можете просто закрыть его и Git Bash вернется в рабочий режим).

Также, если надо записать изменения в этом текстовом редакторе, нужно нажать <esc> <:> <w> (сейчас этого не делайте).

17. Вернулись в Git Bash

```
MINGW64:/d/ProGit/branches

    Коммит в ветке testing

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит

d103@S-T000012680 MINGW64 /d/ProGit/branches (testing)
$ git checkout master
Switched to branch 'master'

d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git merge teting
merge: teting - not something we can merge

d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git merge testing
Merge made by the 'recursive' strategy.
 README.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ |
```

Слияние было осуществлено с использованием рекурсивной стратегии.

18. Выведите лог. Вы видите перечень того, какие коммиты применялись, последним создан коммит «Merge branch ‘testing’». Это название и можно было поменять в текстовом редакторе, но мы не стали этого делать.

19. Нажмите q для выхода в Git Bash.

Таким образом, Git при слиянии создает новый коммит и этим коммитом отмечает момент, когда это слияние было произведено. Это удобно для того, чтобы в случае необходимости это слияние было легко отменено, и вы могли сдвинуться на момент до него.

Итак, **в Git создание и слияние веток – это основная штатная операция. Если вы хотите внести какие-то изменения в проект, следует создать ветку, внести в ней изменения, протестировать, а затем, если всё нормально – влить в основную, если нет, удалить. Это стандартный метод работы с Git.**

Главный принцип: одна задача – это одна ветка!

Решение конфликтов

Конфликт возникает, если в одном и том же месте (файле) есть разные изменения. Это также штатная ситуация, Git предназначен для разрешения конфликтов.

Конфликт означает, что Git не смог слить версии автоматически и вам необходимо это сделать вручную.

Рассмотрим, что делать в случае возникновения конфликта.

1. Создадим конфликт искусственно. Проверьте, что вы находитесь в ветке master.

2. Создайте в папке branches текстовый файл first.txt и напишите внутри «Это первая строчка.».

3. Сделайте коммит.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git add first.txt

d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git commit -m "Создали файл"
[master d081c15] Создали файл
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 first.txt
```

4. Сделайте ветку «feature/1» (Задача 1) и переключитесь на нее:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git checkout -b feature/1
Switched to a new branch 'feature/1'
```

5. Пусть наша задача №1 – сделать красную кнопку. Откройте файл first.txt и напишите в конце «Красная кнопка». Сохраните файл. Проиндексируйте и закоммитьте изменения в текущей ветке feature/1.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1)
$ git add first.txt

d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1)
$ git commit -m "Создали красную кнопку"
[feature/1 d71c1cc] Создали красную кнопку
 1 file changed, 2 insertions(+)
```

6. Снова переключитесь на ветку master.
7. Создайте ветку под названием feature/2 и перейдите на неё.
8. Вторая задача – сделать две зелёных кнопки. Откройте файл first.txt, там в данной ветке должна быть только одна строка. Напишите во второй строке «Тут две зелёных кнопки».
9. Создайте коммит:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/2)
$ git add first.txt

d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/2)
$ git commit -m "Создали две зеленых кнопки"
[feature/2 f3fc0e9] Создали две зеленых кнопки
 1 file changed, 2 insertions(+)
```

Итак, у нас есть две ветки, у которых разошлась история.

10. Переключитесь на первую задачу:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/2)
$ git checkout feature/1
Switched to branch 'feature/1'
```

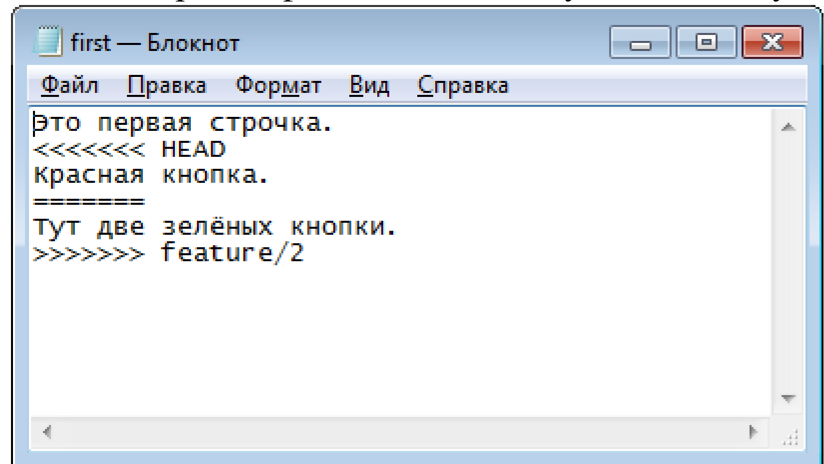
11. Попробуем в эту ветку влить изменения из второй.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1)
$ git merge feature/2
Auto-merging first.txt
CONFLICT (content): Merge conflict in first.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Мы видим надпись CONFLICT – в файле first.txt возник merge-конфликт. Автоматическое слияние невозможно. Нужно поправить конфликты и закоммитить результат.

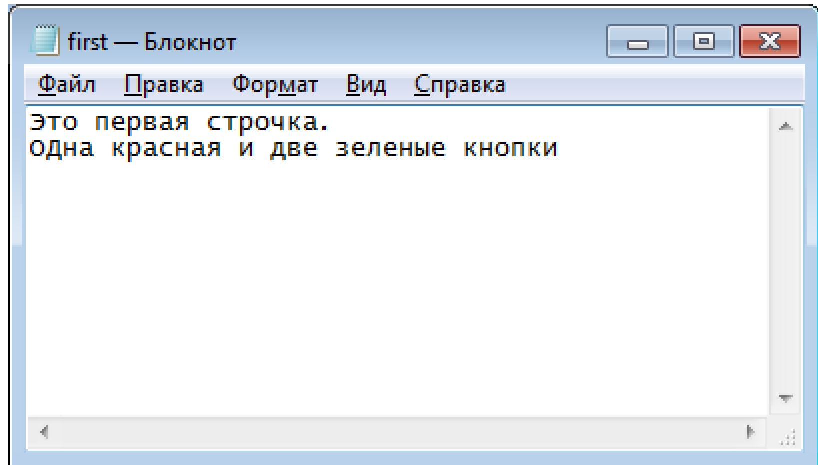
То есть, Git попытался применить последовательную стратегию слияния, обнаружил противоречия и вывел такое сообщение.

12. Откройте файл first.txt и вы увидите следующую картину:



13. В файле появились метки, которые указывают, что же вызвало конфликт, что именно вызывает противоречие, а именно: Красная кнопка - Две зеленые кнопки.

14. Git хочет, чтобы мы оставили что-то одно, требует выбрать некий результат вручную (или оба). Сотрите всю служебную секцию и напишите «Одна красная и две зеленые кнопки»:



Закройте и сохраните файл.

15. Проверьте статус репозитория.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1|MERGING)
$ git status
On branch feature/1
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:   first.txt

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

16. Проиндексируйте файл и закоммитьте его под именем «Конфликт решён».

ДОМАШНЯЯ РАБОТА

1. Начните новый репозиторий на Гитхабе

2. Пользуясь своими знаниями HTML и CSS, создайте в нем файл index.html Пусть в этом файле будет информация о вас, профессиональный профиль. (достаточно пары абзацев, суть ДЗ не в том, чтобы красиво сверстать страницу). Зафиксируйте изменения.

3. Представьте теперь, что руководство ставит задачу: разместить на странице кнопку "Подписаться на новости"

- Создайте новую ветку для выполнения этой задачи

- Добавьте кнопку. Зафиксируйте изменения. Передайте их на гитхаб и убедитесь, что вы видите новую ветку в интерфейсе Гитхаба.

- Влейте изменения из ветки задачи в ветку master - вы должны увидеть merge методом fast-forward

- Передайте все изменения в удаленный репозиторий

4. Поступили сразу две новых задачи: изменить цвет кнопки из пункта 3. и добавить на страницу форму входа через социальные сети (условно, конечно!)

- Создайте для каждой задачи свои ветки
- Выполните их
- Влейте изменения в master
- Передайте изменения в Github

5. Смоделируйте возникновение конфликта, внося изменения в одно и то же место своего файла. Решите конфликт.

Контрольные вопросы:

1. Какая ветка создается сразу после создания репозитория?
2. Как задать новую ветку?
3. Что такое head и какая взаимосвязь между git checkout?
4. Как произвести слияние веток?
5. Что значит fast-forward?

Список литературы:

1. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
2. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

Лабораторная работа №7. Работа с GitHub.

Цель работы:

Регистрация на Github. Создание репозитория, клонирование. Команды Github'a.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

В первую очередь надо установить клиент git: обязательно потребуется консольный клиент, доступный по ссылке http://git-scm.com/downloads (поддерживаются основные ОС), графический клиент можно установить по желанию, исходя из своих предпочтений. На Unix системах можно воспользоваться менеджером пакетов (yum на fedora и подобных или apt-get на debian, ubuntu и подобных) вместо того, чтобы скачивать установщик с сайта.

Далее работа с git будет объясняться на примере работы с консольным клиентом по следующим причинам:

- Чтобы у вас складывалось понимание происходящего и при возникновении проблем вы могли четко объяснить, что вы делали, и было видно, что пошло не так.
- Все нажатия кнопок в графических клиентах в итоге сводят к выполнению определнных команд консольного клиента, в то же время возможности графических клиентов ограничены по сравнению с консольным
- У тех, кто будет работать в классе на стоящих там компьютерах, не будет другого выбора, кроме как пользоваться консольным клиентом (на сколько мне известно, никаких графических клиентов для git там не установлено)

Практическая часть.

В данной работе рассмотрим использование для этих целей популярного в настоящее время сервиса Github.

GitHub — крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки. Веб-сервис основан на системе контроля версий Git и разработан на RubyonRails и Erlang компанией GitHub, Inc. Википедия

На его примере разберем, как работать с получением изменений и их отправкой.

Регистрация на GitHub

GitHub — веб-сервис, который основан на системе Git. Это такая социальная сеть для разработчиков, которая помогает удобно вести коллективную разработку IT-проектов. Здесь можно публиковать и редактировать свой код, комментировать чужие наработки, следить за новостями других пользователей. Именно в GitHub работаем мы, команда Академии, и студенты интенсивов.

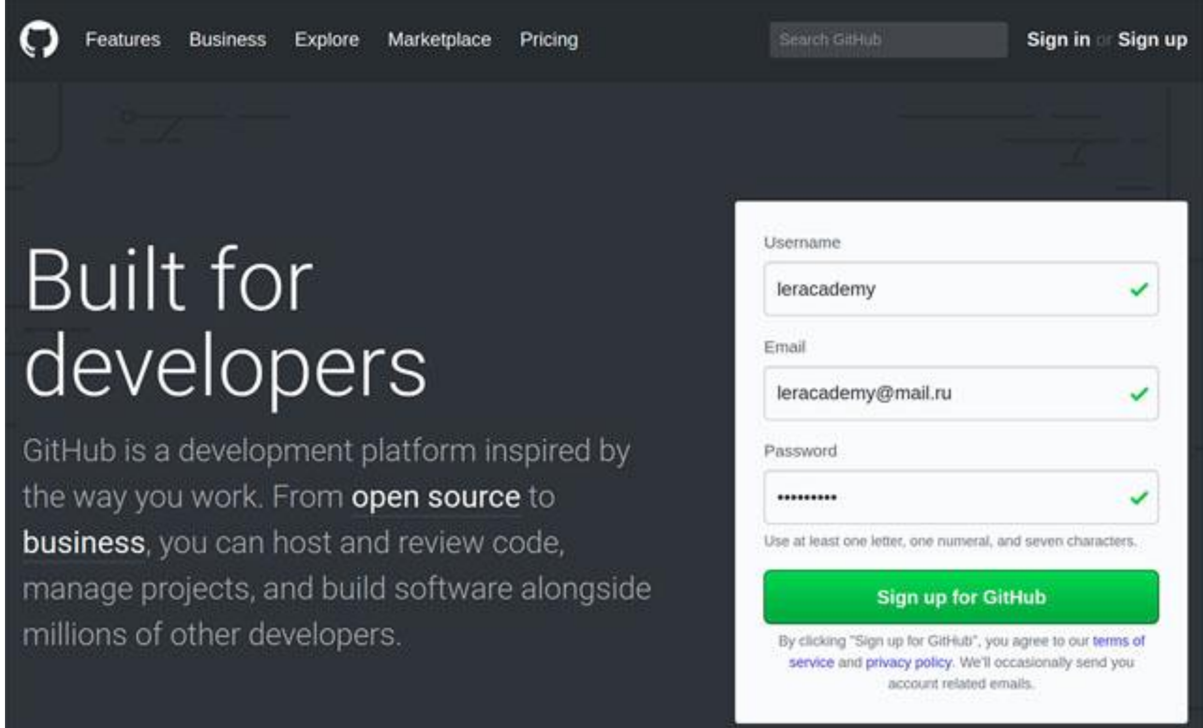
Чтобы начать работу с GitHub, нужно зарегистрироваться на сайте, если вы ещё этого не сделали. За дело.

1. Переходим на сайт GitHub.

Стартовая страница GitHub.

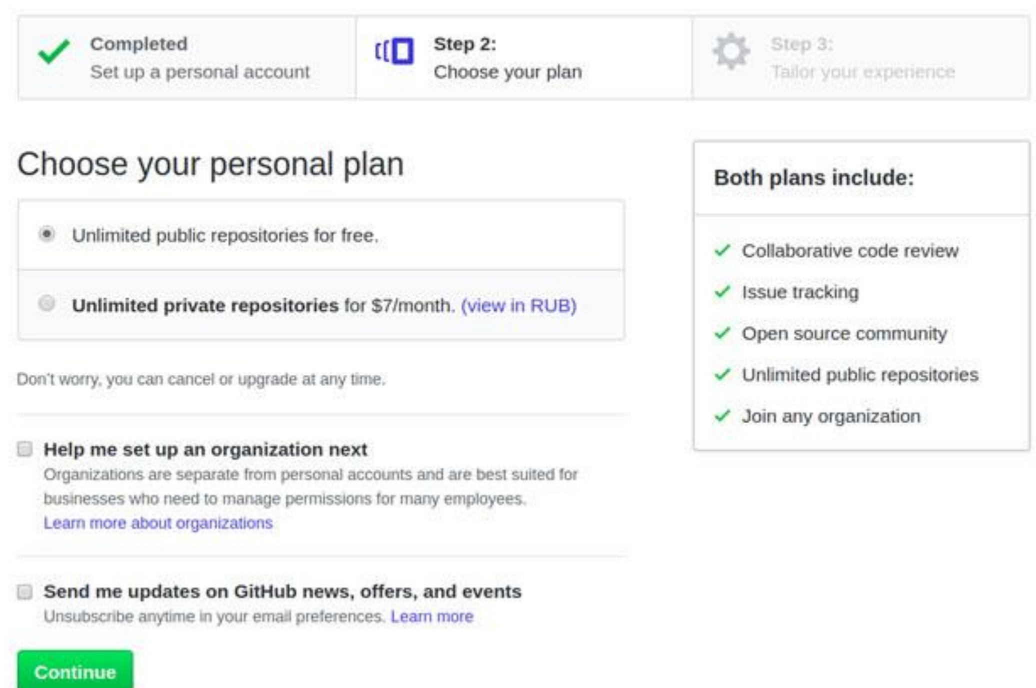
2. Есть два варианта начала регистрации:

- Нажимаем кнопку Sign up (зарегистрироваться), попадаем на страницу регистрации, где вводим обязательные данные: имя пользователя, адрес электронной почты и пароль. После заполнения полей нажимаем Createan account (создать аккаунт).
- Сразу вводим имя, почту и пароль на главной странице GitHub и нажимаем Signup forGitHub (зарегистрироваться на GitHub).



Первый шаг регистрации профиля на стартовой странице GitHub.

3. На втором этапе нужно выбрать тарифный план. GitHub — бесплатный сервис, но предоставляет некоторые платные возможности. Выбираем тарифный план и продолжаем регистрацию.



Выбор тарифа на втором шаге регистрации.

Пользование GitHub бесплатно до тех пор, пока ваш код открыт для всех. Платные версии позволяют создавать закрытые, приватные репозитории, которые видите только вы.

4. Третий шаг — небольшой опрос от GitHub, который вы можете пройти, заполнив все поля и нажать Submit или пропустить, нажав skipthisstep.

Completed
Set up a personal account

Step 2:
Choose your plan

Step 3:
Tailor your experience

How would you describe your level of programming experience?

Very experienced
Somewhat experienced
Totally new to programming

What do you plan to use GitHub for? (check all that apply)

Design
Development
School projects
Project Management
Research
Other (please specify)

Which is closest to how you would describe yourself?

I'm a professional
I'm a student
I'm a hobbyist
Other (please specify)

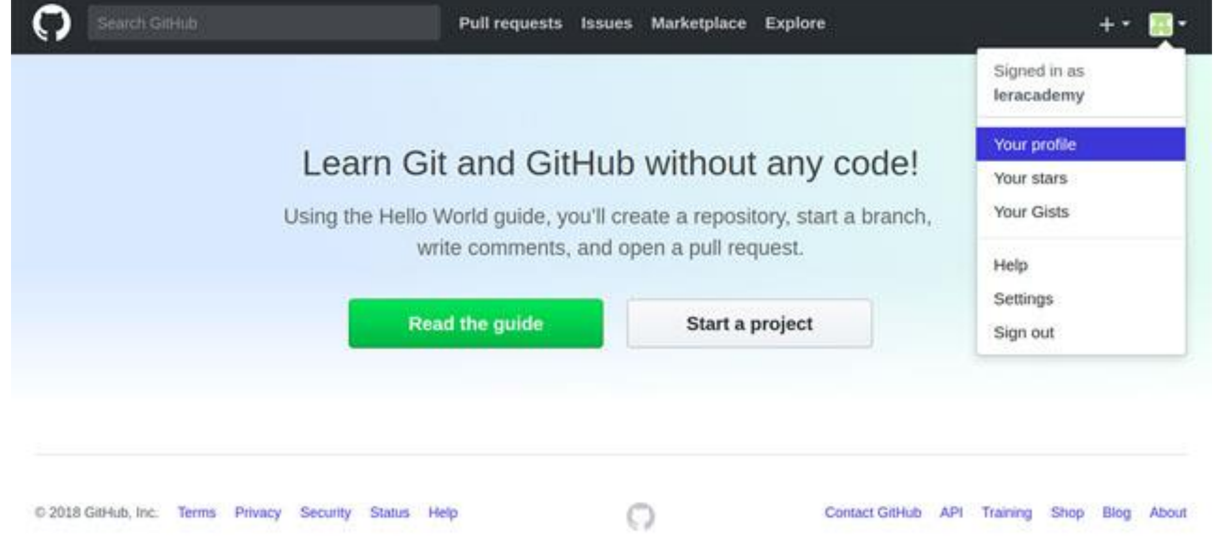
What are you interested in?

e.g. tutorials, android, ruby, web-development, machine-learning, open-source

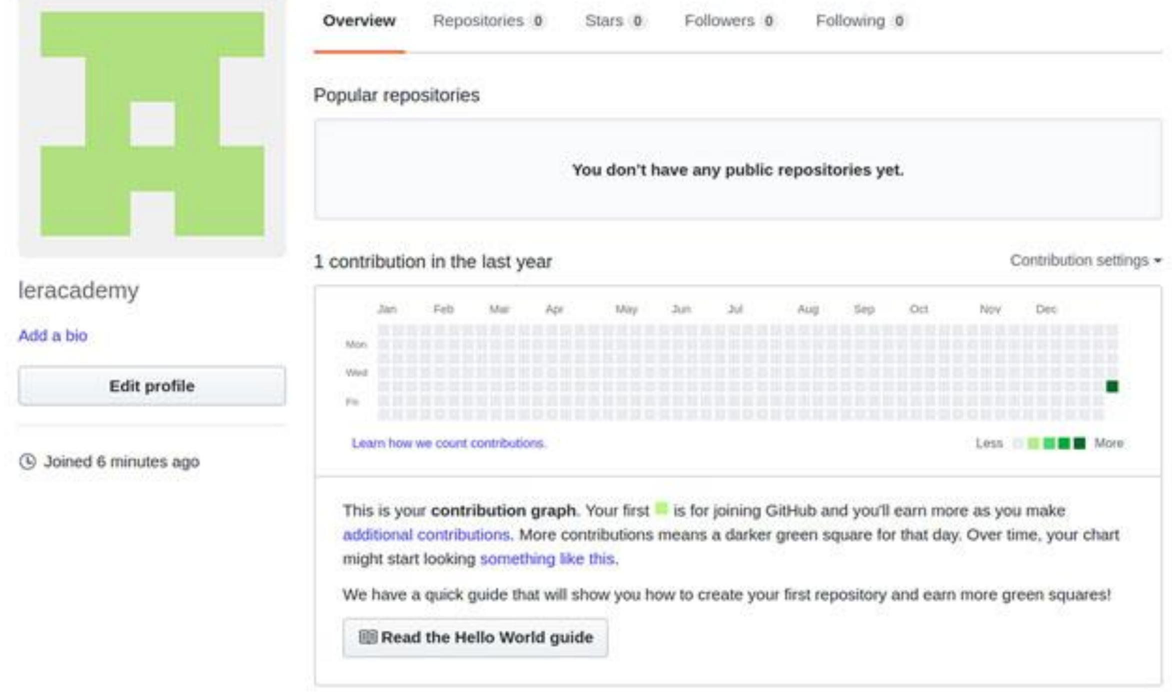
Submit
skip this step

5. Опрос на третьем шаге регистрации.

6. После прохождения всех этапов на сайте, на указанный при регистрации ящик вам придёт письмо от GitHub. Откройте его и подтвердите свой почтовый адрес, нажав Verifyemailaddress (подтвердить электронный адрес) или скопируйте вспомогательную ссылку из письма и вставьте её в адресную строку браузера.



Переход в ваш профиль.



Так выглядит ваш профиль после регистрации.

Теперь у вас есть профиль на GitHub.

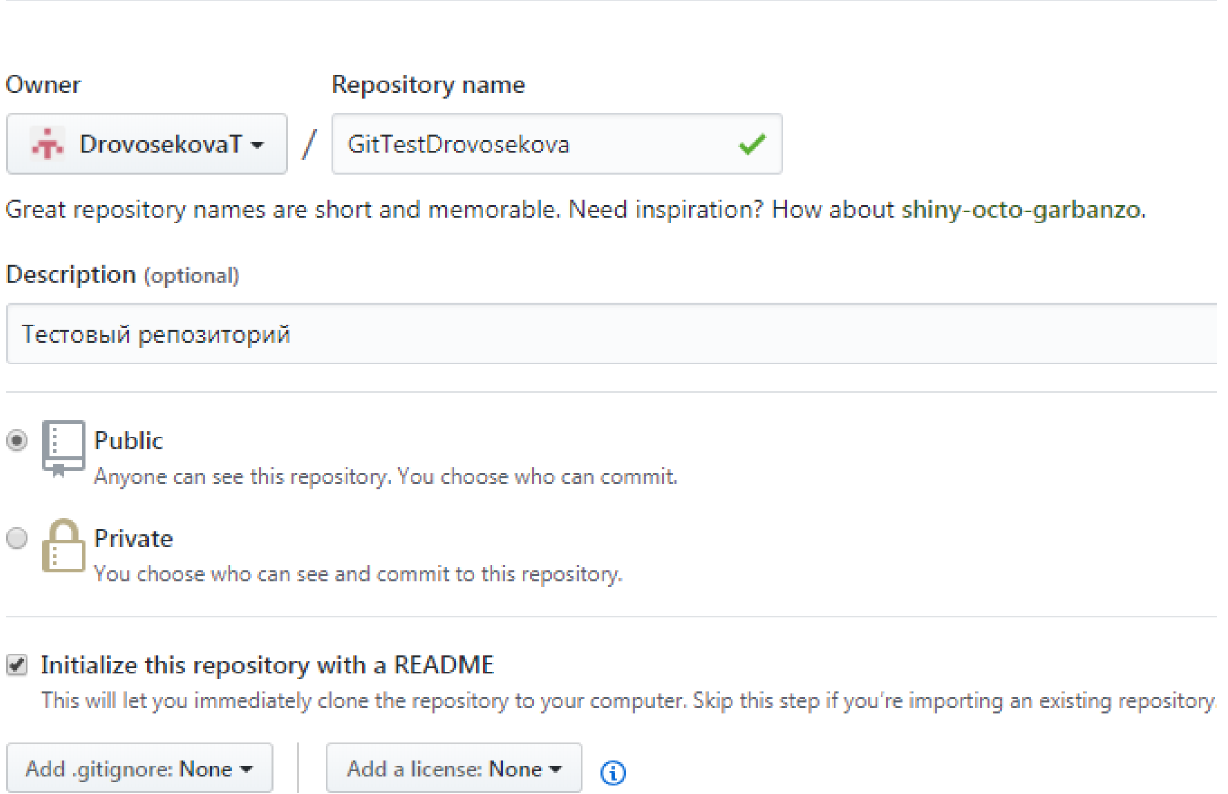
Сейчас у нас нет ни одного репозитория, и мы можем либо создать новый репозиторий, либо ответвиться (fork) от уже существующего чужого репозитория и вести собственную ветку разработки. Затем, при желании, свои изменения можно предложить автору исходного репозитория (Pullrequest).

7. Если Вы находитесь в своем профиле, то в верхней части окна видите один из пунктов меню «Repositories 0». Это означает, что у вас в данный момент нет созданных репозиториев. На этой же вкладке есть кнопка «New», которая позволяет создать новый репозиторий. Нажимаем её.

8. Открылось окно создания репозитория. Даем ему имя (например, gittest) можно написать описание. Он должен быть публичным. Также

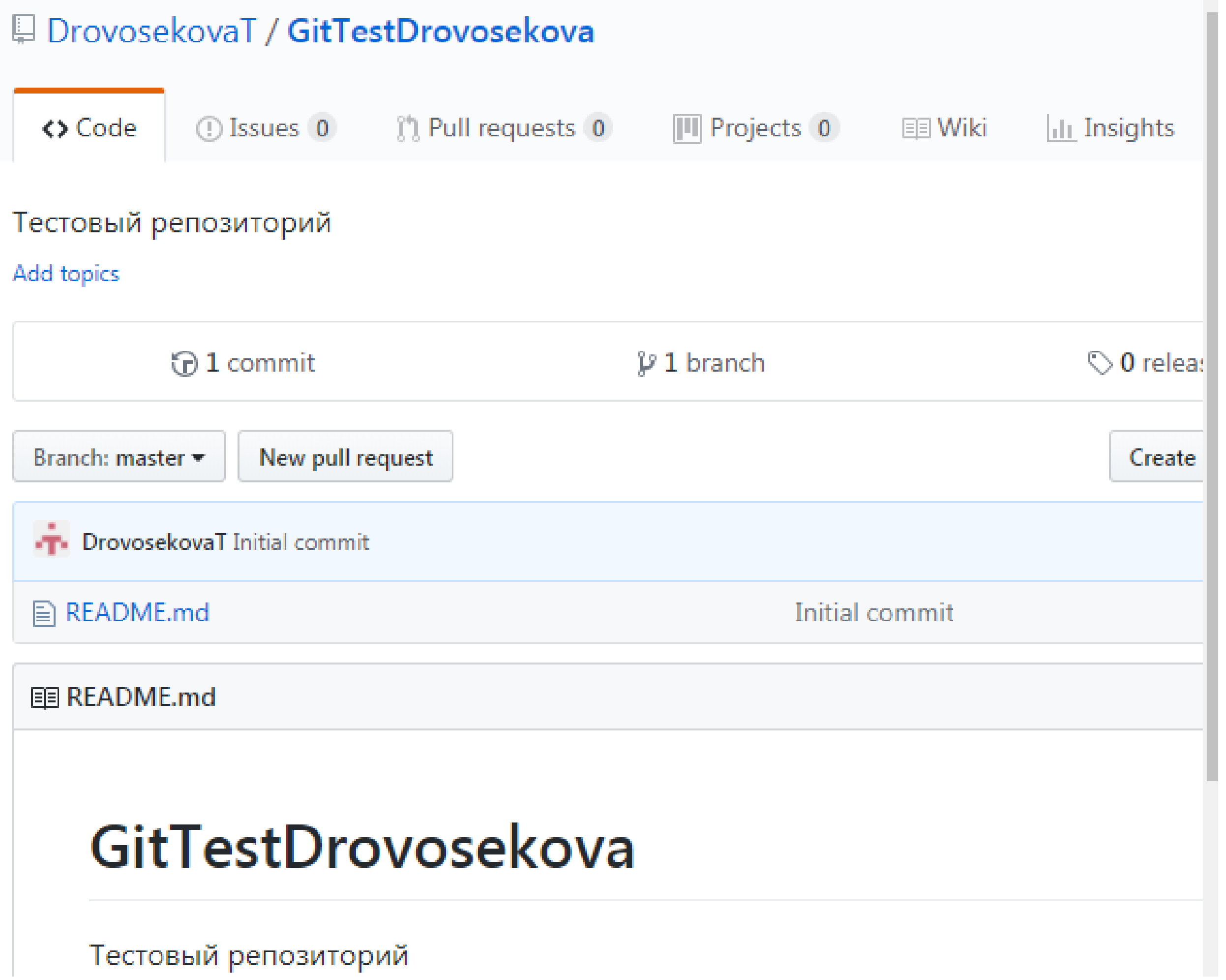
установите галочку там, где предлагается поместить в репозиторий текстовый файл Readme.

Нажмите кнопку «Create repository».

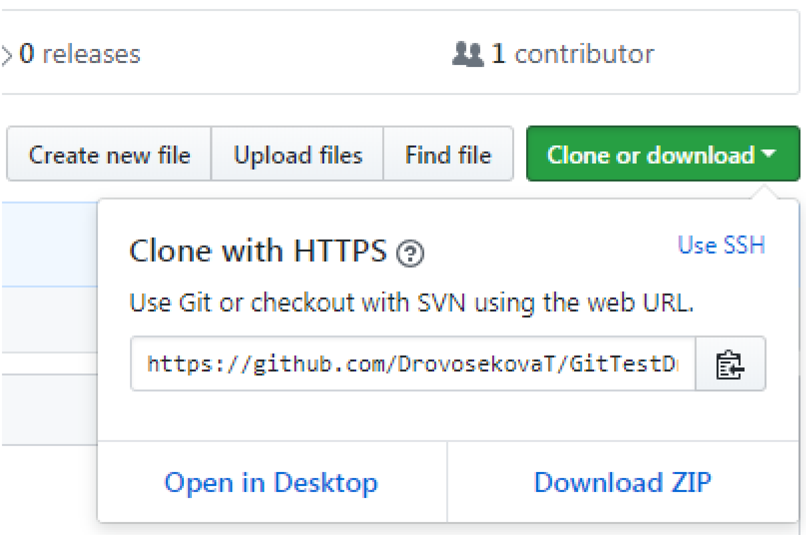


9. Когда вы создаете новый репозиторий таким образом, на Github появляется папка и для нее выполняется команда gitinit, т.е. происходит то же самое, что мы делали в предыдущих лабораторных работах вручную.

10. После создания репозитория вы в него попадаете и видите, что в нем в данный момент находится один файл – README.md, который был создан автоматически:



11. Далее нам нужно скопировать в буфер обмена адрес нашего репозитория на Github. Для этого нажмите кнопку «Cloneordownload»и в открывшемся окне копку справа от адреса ссылки:



12. Откройте консоль GitBash (она должна быть у вас открыта после лабораторной работы 2, если нет – перейдите в папку tmp, с которой мы работали). Даем команду

git clonehttps://github.com/DrovosekovaT/GitTestDrovosekova.git **test**

(вставьте из буфера обмена свой адрес репозитория и после него через пробел название папки, где будет создан клон удаленной папки-репозитория, в данном случае test).

Эта команда создаст в папке test копию репозитория с указанного адреса с Github, автоматически Github подключит к локальному репозиторию как удалённый и назовёт его origin, т.е. оригинал (перейдите в папку test и проверьте это командой gitremote -v), автоматически в нашем локальном репозитории создаст локальную ветку master (проверьте это командой gitstatus), которую свяжет с веткой master в удаленном репозитории на Github.

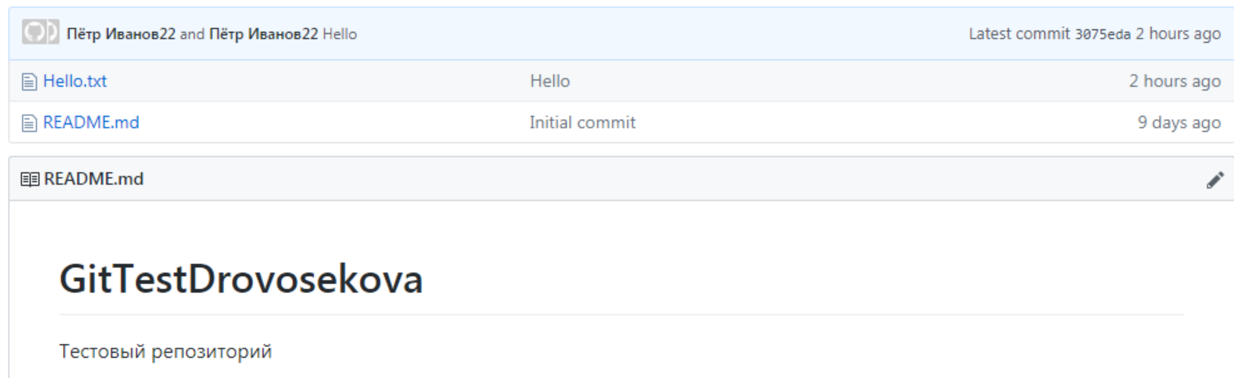
13. Теперь рассмотрим, как синхронизировать локальный репозиторий (клон) с оригиналом, который находится на Github. В клонированной папке test создайте локально файл Hello.txt, напишите в нем текст «Hello!!!», закройте его, сохранив изменения. Теперь у нас репозиторий содержит 2 файла, а на Github только один. Для того, чтобы зафиксировать изменения, выполните команду gitaddHello.txtи создайте коммит:

```
Танюшка@NoteBook MINGW64 /d/tmp/test
 git add Hello.txt

Танюшка@NoteBook MINGW64 /d/tmp/test
 git commit -m "Hello"
[master 3075eda] Hello
 1 file changed, 1 insertion(+)
 create mode 100644 Hello.txt
```

14. Выполните команду **gitpush**. Эта команда берет имеющиеся локальные изменения и отправляет их в удалённый репозиторий. В ответ на запрос введите своё имя на Github и пароль.

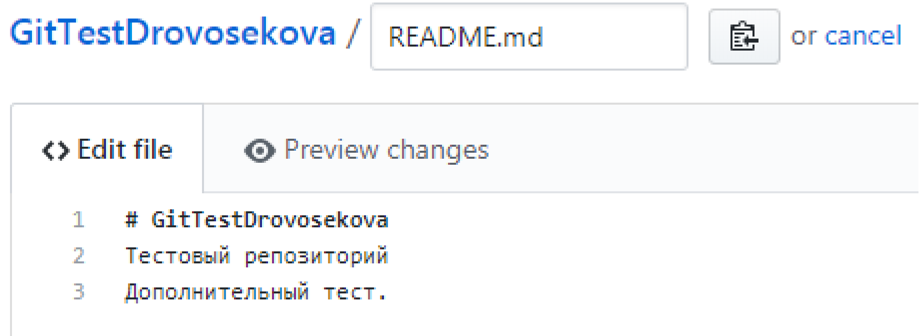
```
Танюшка@NoteBook MINGW64 /d/tmp/test
 git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes | 306.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/DrovosekovaT/GitTestDrovosekova.git
   2519672..3075eda  master -> master

Танюшка@NoteBook MINGW64 /d/tmp/test
 |
```

15. Откройте репозиторий на сайте Github, убедитесь, что файл добавился:



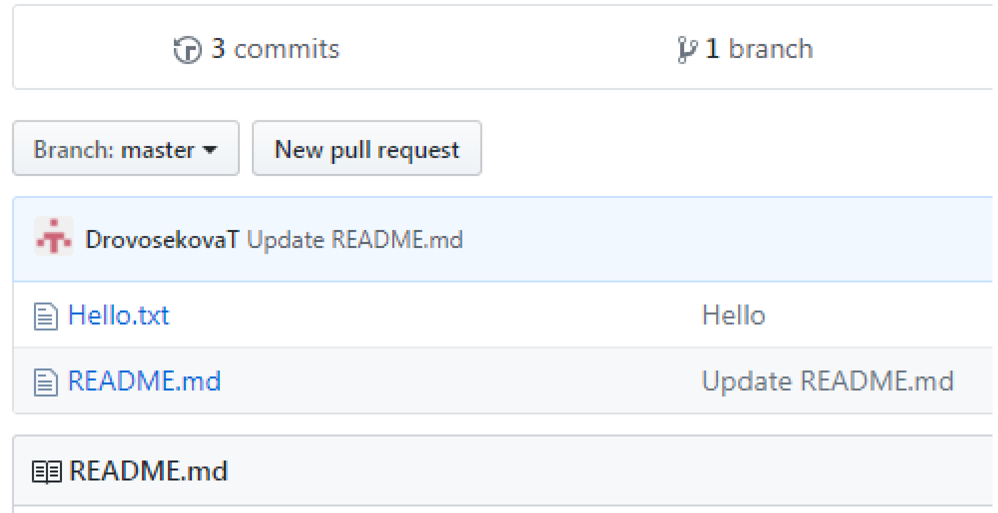
Если вы откроете добавившийся файл Hello.txt, то увидите свой текст. Также вы видите название коммита во втором столбце.

16. Также есть обратная команда. Допустим, у нас на Github появились какие-то изменения, которых нет на локальном компьютере. Для примера откройте и отредактируйте файл README.md. Для этого откройте его на Github и нажмите кнопку с карандашом (Editthisfile), допишите в третьей строке какой-либо текст:



Теперь нажмите кнопку «Commitchanges» в нижней части страницы.

17. Перейдите в папку репозитория, вы увидите, что коммитов стало три.



18. Теперь откройте GitBash на вашем локальном компьютере и вызовите команду gitlog. Вы увидите, что коммитов по-прежнему два. Необходимо загрузить изменения с Github.

19. Введите команду gitpull. Эта команда загрузит изменения из удалённого репозитория.

```
Танюшка@NoteBook MINGW64 /d/tmp/test
 git log
commit 3075eda5bb3f100642e5ed6e3bd2c7f01419b7d0 (HEAD -> master, origin/mas
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 1 13:29:15 2018 +0300

    Hello

commit 2519672015349dcc5622c473733df2f40365ba66
Author: DrovosekovaT <42639140+DrovosekovaT@users.noreply.github.com>
Date:   Thu Aug 23 13:36:27 2018 +0300

    Initial commit

Танюшка@NoteBook MINGW64 /d/tmp/test
 git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/DrovosekovaT/GitTestDrovosekova
   3075eda..ca53ead  master     -> origin/master
Updating 3075eda..ca53ead
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

20. Вызовите gitlog и убедитесь, что коммитов стало три.

```
Танюшка@NoteBook MINGW64 /d/tmp/test
 git log
commit ca53ead5d8d08764a7e3b5e8dd5ea30e1fd77fef (HEAD -> master, origi
Author: DrovosekovaT <42639140+DrovosekovaT@users.noreply.github.com>
Date:   Sat Sep 1 15:20:31 2018 +0300

    Update README.md

commit 3075eda5bb3f100642e5ed6e3bd2c7f01419b7d0
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 1 13:29:15 2018 +0300

    Hello

commit 2519672015349dcc5622c473733df2f40365ba66
Author: DrovosekovaT <42639140+DrovosekovaT@users.noreply.github.com>
Date:   Thu Aug 23 13:36:27 2018 +0300

    Initial commit
```

21. Чтобы посмотреть содержимое обновленного файла, введите команду catREADME.md:

```
Танюшка@NoteBook MINGW64 /d/tmp/test
 cat README.md
# GitTestDrovosekova
Тестовый репозиторий
Дополнительный тест.
```

Примечание: Если у вас уже есть репозиторий на компьютере, то вы можетеего клонировать в пустую (без файла README) папку на Github.

Итоги:

gitpull – не просто «достает изменения» из удаленного репозитория, но и пытается применить изменения из удалённого репозитория к нашему (влить удалённую ветку в локальную). Ветки будут изучаться далее.

gitpush – отправляет наши наборы изменений на удалёный репозиторий.

Таким образом, для чего применяется Github в реальной работе?

1. Githubочень удобно использовать для быстрого бесплатного создания репозитория в интернете.

Для чего может понадобиться репозиторий в интернете?

1. Дать кому-то на него ссылку, чтобы этот человек мог удобно посмотреть на код, хранящийся там, удобно посмотрел на историю изменений.
2. Githubможет служить посредником для коллективной работы.
3. Если вы работаете за несколькими компьютерами, можно клонировать на них репозиторий с Github и синхронизировать через него все версии.

САМОСТОЯТЕЛЬНАЯ РАБОТА

Git, как распределенная система

Мы узнали, что такое Git. Но почему все-таки это распределенная или, как еще говорят, децентрализованная система? Узнаем на уроке!

- понятие удаленного репозитория
- настройка связи между репозиториями
- команды push и pull
- команда fetch

Кроме того мы научимся пользоваться сервисом GitHub и создадим на нем учебный проект

ДОМАШНЯЯ РАБОТА

1. Создайте у себя на компьютере еще один репозиторий ("второй"). Пустой. Свяжите его с уже существующим ("первым") с помощью git remote. Передайте в него изменения из первого. Объясните полученный результат.

2. Создайте на гитхабе пустой репозиторий ("А"). ВНИМАНИЕ! Не создавайте в нем файл readme!

- Установите его в качестве удаленного для репозитория "первый"
- Передайте из "первого" в "А" изменения
- Приложите ссылку на "А" к своему ДЗ

3. Создайте на гитхабе новый непустой (с файлом readme) репозиторий "Б". Склонируйте его к себе ("третий"). Выполните локально несколько коммитов, передайте их в "Б". Приложите ссылку на "Б"

4.* Сделайте еще один клон "Б" - "четвертый". Измените один и тот же файл в локальном репозитории "третий", git push, затем в "четвертом" (те же строки, но другие изменения) - и тоже попытка git push. Если возникнет конфликт - попробуйте объяснить его суть.

Контрольные вопросы:

1. Как отправить изменения на удаленный репозиторий?
2. Как влить изменения в локальную ветку из удаленного репозитория?
3. Какой командой можно зафиксировать изменения в репозитории?
4. Как используется Gihub в реальной работе?
5. Опишите принцип коллективной работы с Github

Список литературы:

1. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
2. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

Лабораторная работа №8. Знакомство с Rational Rose

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

UML диаграммы в Rational Rose

Rational Rose – мощное CASE-средство для проектирования программных систем любой сложности. Одним из достоинств этого программного продукта будет возможность использования диаграмм на языке UML. Можно сказать, что Rational Rose является графическим редактором UML диаграмм.

CASE-средство (Computer Aided Software Engineering) – это инструмент, который позволяет автоматизировать процесс разработки информационной системы и программного обеспечения.

UML (Unified Modeling Language) – это графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке программных систем. В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название **диаграмм**.

В распоряжение проектировщика системы Rational Rose предоставляет следующие типы диаграмм, последовательное создание которых позволяет получить полное представление о всей проектируемой системе и об отдельных ее компонентах:

- Use case diagram (диаграммы прецедентов);
- Deployment diagram (диаграммы топологии);
- Statechart diagram (диаграммы состояний);
- Activity diagram (диаграммы активности);
- Interaction diagram (диаграммы взаимодействия);
- Sequence diagram (диаграммы последовательностей действий);
- Collaboration diagram (диаграммы сотрудничества);
- Class diagram (диаграммы классов);
- Component diagram (диаграммы компонент).

Интерфейс Rational Rose

После запуска программы Rational Rose автоматически создается новый проект и в рабочем окне диаграммы появляется по умолчанию окно диаграммы классов.

Рабочий интерфейс Rational Rose состоит из различных элементов, основными из которых являются:

- Главное меню программы;
- Окно диаграммы;
- Стандартная панель инструментов;
- Окно документации;
- Окно браузера;
- Окно журнала;
- Специальная панель инструментов.

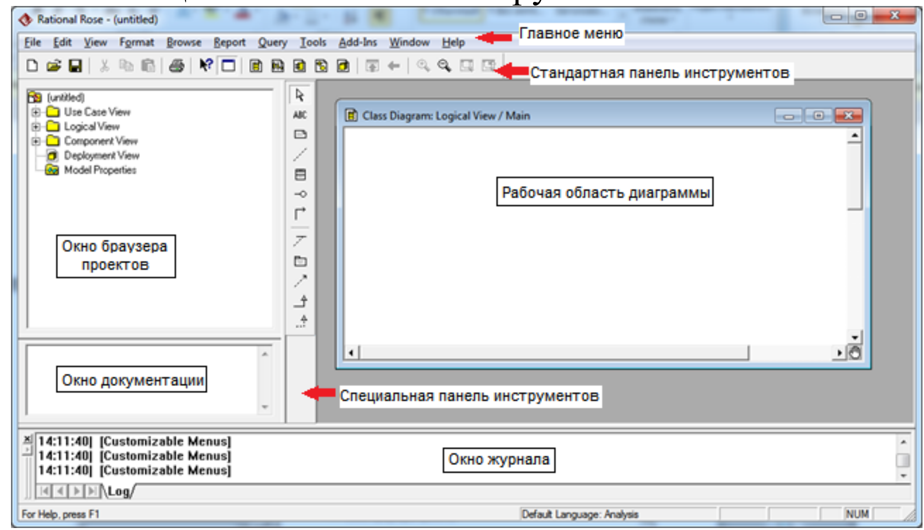


Рисунок 4.1 – Общий вид рабочего интерфейса CASE-средства Rational Rose

Практическая часть.

Отдельные пункты **главного меню**, назначение которых понятно из их названий, объединяют сходные операции, относящиеся ко всему проекту в целом. Некоторые из пунктов меню содержат хорошо знакомые функции (открытие проекта, вывод печать диаграмм, копирование в буфер и вставка из буфера различных элементов диаграмм). Другие настолько специфичны, что могут потребовать дополнительных усилий на изучение (опции генерации программного кода, проверка согласованности моделей, подключение дополнительных модулей).

Стандартная панель инструментов обеспечивает быстрый доступ к тем командам меню, которые выполняются разработчиками наиболее часто. Пользователь может настроить внешний вид этой панели по своему усмотрению.

Окно браузера предназначено для представления модели в виде иерархической структуры, которая упрощает навигацию и позволяет отыскать любой элемент модели в проекте. При этом любой элемент, который разработчик добавляет в модель, сразу отображается в окне браузера. Соответственно, выбрав элемент в окне браузера, мы можем его

визуализировать в окне диаграммы или изменить его спецификацию. Браузер позволяет также организовывать элементы модели в пакеты и перемещать элементы между различными представлениями модели.

Специальная панель инструментов по умолчанию предназначена для построения диаграммы классов модели. Состав панели можно настраивать под конкретный вид диаграммы.

Окно диаграммы является основной рабочей областью ее интерфейса, в которой визуализируются различные представления модели проекта. При разработке нового проекта окно диаграммы представляет собой чистую область, не содержащую никаких элементов модели. Одновременно в окне диаграммы могут присутствовать несколько диаграмм, однако активной может быть только одна из них.

Окно документации предназначено для документирования элементов представления модели. В него можно записывать самую различную информацию, в том числе и на русском языке. Эта информация в последующем преобразуется в комментарии и никак не влияет на логику выполнения программного кода. В окне документации отображается только та информация, которая относится к отдельному выделенному элементу диаграммы.

Окно журнала предназначено для автоматической записи различной служебной информации, образующейся в ходе работы с программой. В журнале фиксируется время и характер выполняемых разработчиком действий, таких как обновление модели, настройка меню и панелей инструментов, а также сообщения об ошибках, возникающих при генерации программного кода.

Контрольные вопросы:

1. Для чего нужен программный продукт Rational Rose?
2. Что такое UML?
3. Что такое CASE-средство?
4. Как называются графические конструкции, используемые в UML?
5. Опишите пункты главного меню и их предназначение

Список литературы:

1. Михеева, Е. В. Информационные технологии в профессиональной деятельности :учеб.пособие / Е.В. Михеева. - 14-е изд., стер. - М. : Академия, 2016. - 384 с.
2. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.

Лабораторная работа №9. Диаграмма претендентов

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Use case diagram (диаграмма прецедентов)

Этот вид диаграмм позволяет создать список операций, которые выполняет система. Часто этот вид диаграмм называют диаграммой функций, потому что на основе набора таких диаграмм создается список требований к системе и определяется множество выполняемых системой функций.

Каждая такая диаграмма или, как ее обычно называют, каждый Use case – это описание сценария поведения, которому следуют действующие лица (Actors).

Данный тип диаграмм используется при описании бизнес процессов автоматизируемой предметной области, определении требований к будущей программной системе. Отражает объекты как системы, так и предметной области, и задачи, ими выполняемые.

При построении диаграммы прецедентов будем использовать пиктограммы типа "прецеденты" и "актеры".

Практическая часть:

Прецедент – это описание множества последовательных событий, выполняемых компьютерной системой, которые приводят к наблюдаемому актером результату. Графически прецедент изображается в виде ограниченного непрерывной линией эллипса, обычно содержащего только имя прецедента.

Актер – это кто-то (или что-то) внешний по отношению к компьютерной системе, кто взаимодействует с ней. Графически актер изображается в виде пиктограммы, представляющей человека, поскольку актер — это человек или группа людей, использующих данные, предоставляемые компьютерной системой.

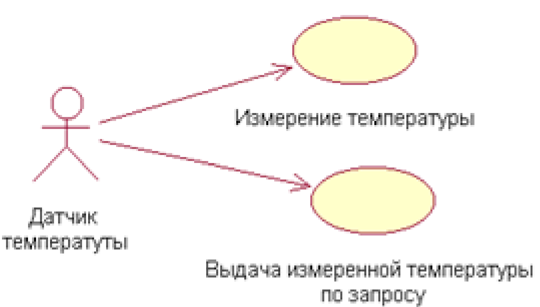


Рисунок 4.2 – Пример диаграммы прецедентов

Class diagram (диаграммы классов)

Этот вид диаграмм позволяет создавать логическое представление системы, на основе которого создается исходный код описанных классов. Диаграмма классов служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования и содержит детальную информацию об архитектуре программной системы.

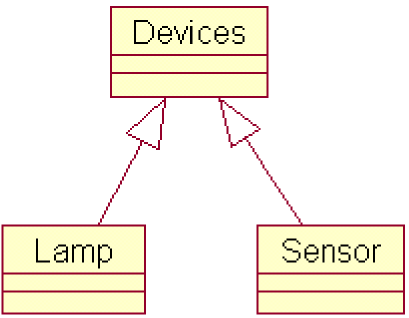


Рисунок 4.3 – Пример диаграммы классов

Контрольные вопросы:

1. Какие атрибуты классов вам известны?
2. Как задать аргументы операции на диаграмме классов?

Список литературы:

3. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
4. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

Лабораторная работа №10. Отношения на диаграммах

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Между компонентами диаграмм могут существовать различные отношения, которые описывают их взаимодействие. В языке UML имеется несколько стандартных видов отношений, например, ассоциация и обобщение.

Практическая часть:

Ассоциация – это структурное двунаправленное отношение, описывающее совокупность взаимоотношений между объектами. Ассоциация может иметь имя и кратность. Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Наиболее распространенными являются следующие формы записи кратности отношения ассоциации:

1. Целое неотрицательное число (включая цифру 0). Предназначено для указания кратности, которая является строго фиксированной для элемента соответствующей ассоциации.
2. Два целых неотрицательных числа, разделенные двумя точками и записанные в виде: "первое число .. второе число", например, 1..5. Очевидно, что первое число должно быть строго меньше второго числа в арифметическом смысле, при этом первое число может быть равно 0.

Если кратность отношения ассоциации не указана, то по умолчанию принимается ее значение, равное 1.

Обобщение – это однонаправленное отношение, называемое "потомок/прародитель", в котором объект "потомок" может быть подставлен вместо объекта прародителя (родителя или предка). Потомок наследует структуру и поведение своего родителя. Графически обозначается сплошной линией со стрелкой в форме незакрашенного треугольника. Стрелка всегда указывает на родителя.

Контрольные вопросы:

1. Какие элементы содержит специальная панель инструментов для создания диаграммы классов?
2. Как добавить элемент на специальную панель инструментов?

Список литературы:

1. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
2. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

Лабораторная работа №11. Диаграмма прецедентов проекта

Цель работы:

Изучить сценарий проекта, определить действующие лица системы и сценарии их поведения.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Сценарий проекта регистрации учебных курсов

Сначала каждый преподаватель университета заполняет специальную форму, в которой указывает, какие учебные курсы он намерен вести в следующем семестре. Данные из формы помещаются в университетский компьютер работником регистратуры.

После этого из полученных данных формируется каталог курсов, который раздается студентам. Студенты выбирают из каталога те курсы, на которых они собираются учиться, и подают заявки на обучение в регистратуру. Все эти данные также попадают в компьютер, где происходит их обработка и формирование списков курсов и студентов. В задачи создаваемой системы входит, в частности, такое комплектование учебных курсов, чтобы каждый курс посещало бы от трех до десяти студентов. Если на какой-то курс не набирается трех студентов, он отменяется.

После формирования курсов преподаватели получают списки студентов, которых им предстоит обучать, а каждый студент получает подтверждение о зачислении на курс и счет на оплату.

Практическая часть:

1.2. Определить действующие лица системы.

Согласно сценарию действующих лиц, в создаваемой системе четыре: преподаватель, студент, регистратор и биллинговая программа – система оплаты. Первые три выбраны действующими лицами, поскольку они активно взаимодействуют с создаваемой системой. Биллинговая же программа чаще всего является отдельным программным продуктом, а в нашем случае она получает информацию для своей работы от создаваемой курсовой системы, поэтому может считаться самостоятельным действующим лицом.

1.3. Определить сценарии поведения действующих лиц. Каждый сценарий описывает некоторое требование к функциям системы:

- Выбор курсов для преподавателя;
- Запрос расписания курсов;

- Регистрация на курсы;
- Создание каталогов ресурсов;
- Хранение информации о курсах;
- Хранение информации о преподавателях;
- Хранение информации о студентах.

1.4. Запустить программу Rational Rose. Ярлык программы: . При запуске в окне **Create New Model** нажать кнопку **Cancel**.

1.5. В программе Rational Rose создать пустой проект. Для этого:

1.5.1. Переключиться на папку **Use Case View** – для работы с диаграммами прецедентов (Use case) и открыть контекстное меню нажатием правой кнопки мыши. Если теперь выбрать пункт **New → Actor** (рис. 4.4), то вы получите действующее лицо, которому следует дать имя **Преподаватель**.

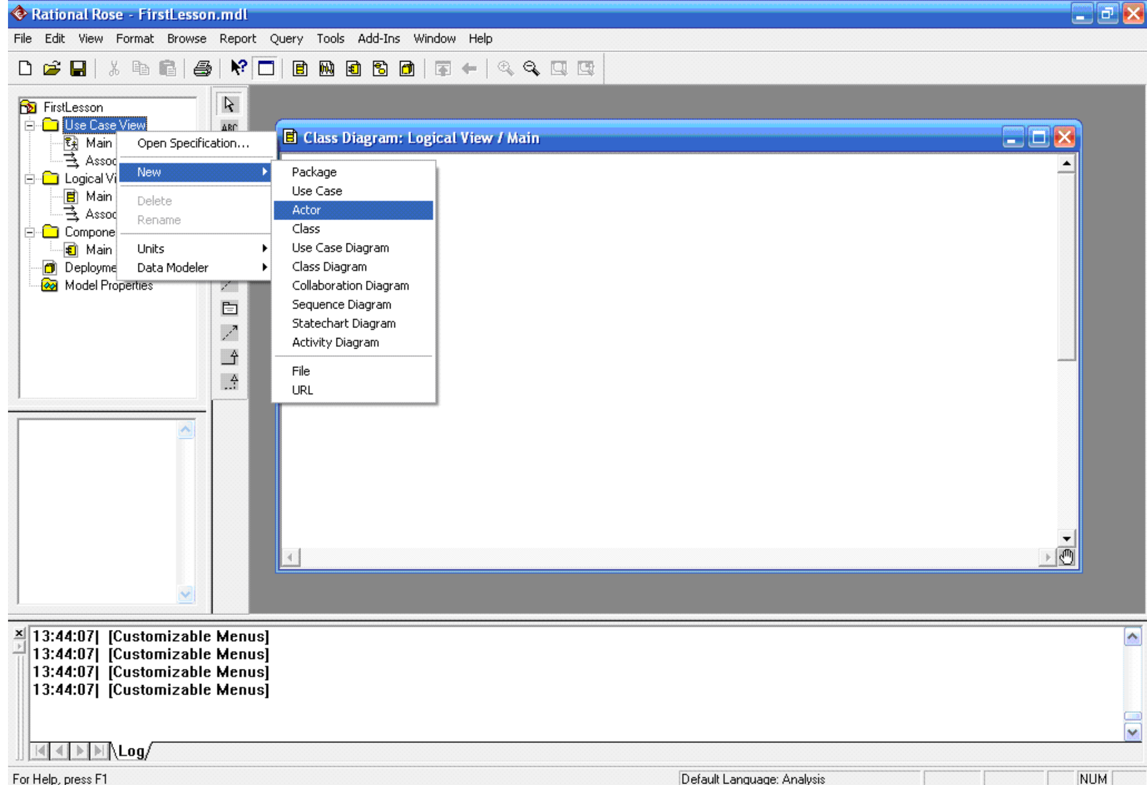


Рисунок 4.4 – Создание действующего лица

Далее аналогичным образом необходимо создать всех действующих лиц системы (рис. 4.5).

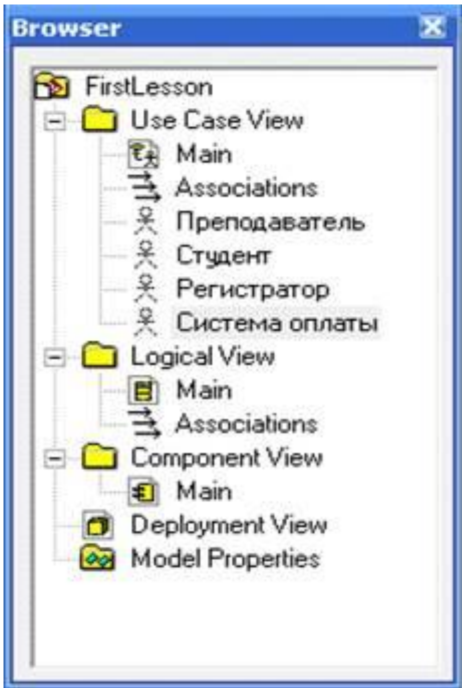


Рисунок 4.5 – Вид окна браузера проекта после создания всех действующих лиц

1.5.2. С помощью контекстного меню папки **Use Case View** и команды **New → Use Case** создать сценарий поведения действующих лиц, перечисленные в п. 3.

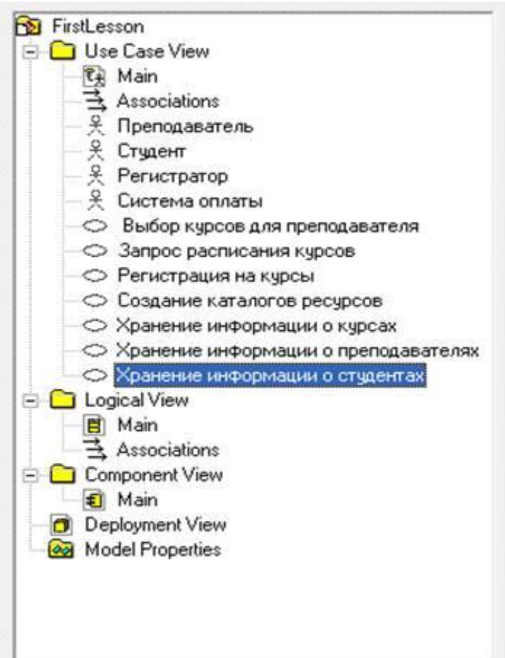


Рисунок 4.6 – Вид окна браузера проекта после создания всех сценариев поведения

Контрольные вопросы:

1. Как добавить на диаграмму последовательности действующее лицо, объект и сообщение?

2. Как производится соотнесение сообщений с операциями на диаграммах последовательности?

Список литературы:

1. Михеева, Е. В. Информационные технологии в профессиональной деятельности :учеб.пособие / Е.В. Михеева. - 14-е изд., стер. - М. : Академия, 2016. - 384 с.
2. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.

Лабораторная работа №12. Диаграмма сценариев поведения

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Каждая такая диаграмма или, как ее обычно называют, каждый Use case – это описание сценария поведения, которому следуют действующие лица (Actors).

Данный тип диаграмм используется при описании бизнес процессов автоматизируемой предметной области, определении требований к будущей программной системе. Отражает объекты как системы, так и предметной области и задачи, ими выполняемые.

Практическая часть.

1.6. Построить диаграмму сценариев поведения (прецедентов). Для этого:

1.6.1. Открыть окно построения диаграммы прецедентов двойным щелчком на пиктограмме **Main** из папки **Use Case View**.

1.6.2. Из окна браузера перетащить в окно построения диаграммы прецедентов все созданные действующие лица и сценарии поведения (рис. 4.7).

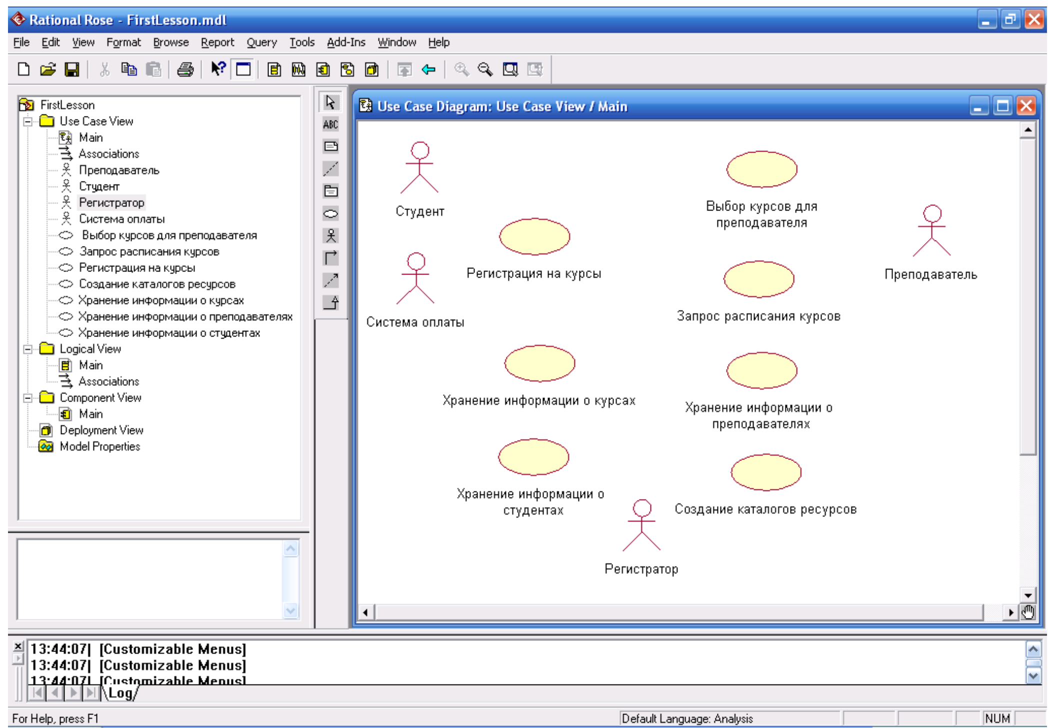


Рисунок 4.7 – Диаграмма сценариев поведения без связей

1.6.3. Установить взаимосвязи между действующими лицами и сценариями поведения (рис. 5.8). Для этого на специальной панели инструментов выбрать тип связи **Однонаправленная ассоциация (Unidirectional Association)** , после чего протянуть линию между действующим лицом и сценарием поведения. В результате на диаграмме возникнет стрелка.

1.6.4. Создать границы между актерами и прецедентами, воспользовавшись пунктом меню **Tools → Create → Note Anchor** или пиктограммой специальной панели инструментов **Anchor Note to Item**. Выбрав инструмент необходимо щелкнуть один раз на середине одной из стрелок отношения, далее во всех углах интерфейса и завершить границу в месте начала ее рисования. Если прямоугольник получился не очень ровный то это можно исправить, выбрав в меню **Format → Line Style → Rectilinear**.

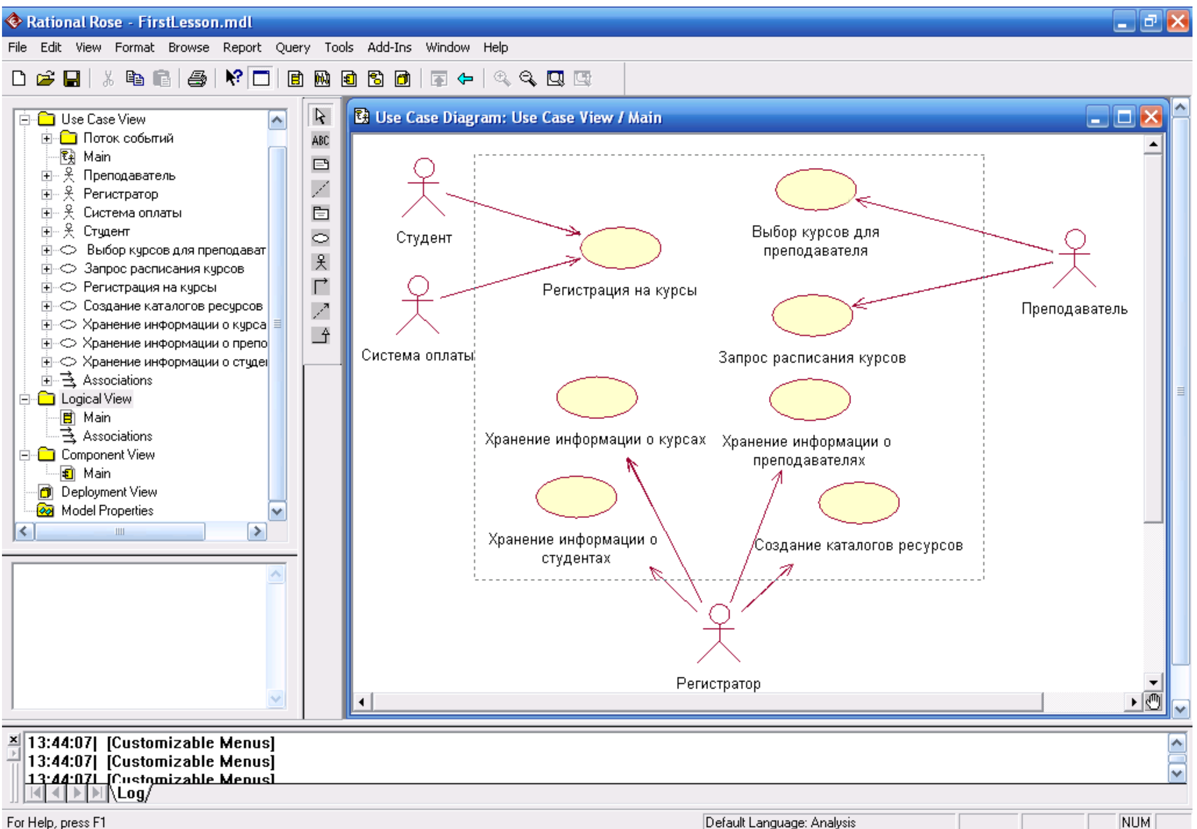


Рисунок 4.8 – Окончательный вариант диаграммы прецедентов системы регистрации учебных курсов

Контрольные вопросы:

1. Какие элементы содержит специальная панель инструментов для создания диаграммы деятельности?
2. Как задать вопрос условия перехода для элемента ветвления?

Список литературы:

1. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
2. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

Лабораторная работа №13. Диаграмма классов концептуальной схемы базы данных

<u>*Цель работы:*</u>

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

<u>*Компетенции:*</u>

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Диаграмма классов (англ. Static Structure diagram) — структурная диаграмма языка моделирования UML, демонстрирующая общую структуру иерархии классов системы, их коопераций, атрибутов (полей), методов, интерфейсов и взаимосвязей между ними. Широко применяется не только для документирования и визуализации, но также для конструирования посредством прямого или обратного проектирования.

Практическая часть:

1) раскрыть папку **Logical View** в браузере проекта и дважды щелкнуть на пиктограмме **Main**;

2) выполнить команды главного меню: **Browse → Class Diagram**.

2.2. В окне построения диаграммы классов создать четыре класса:

- Пользователь;
- Преподаватель;
- Студент;
- Учебный курс.

Для этого можно воспользоваться:

1) контекстным меню папки **Logical View** в браузере проекта, в котором выбрать команды **New → Class**, а затем перетащить созданный класс из окна браузера проекта в область окна диаграммы классов;

2) кнопкой **Class** на специальной панели инструментов, после нажатия которой щелкнуть левой кнопкой мыши на свободном месте окна диаграммы классов.

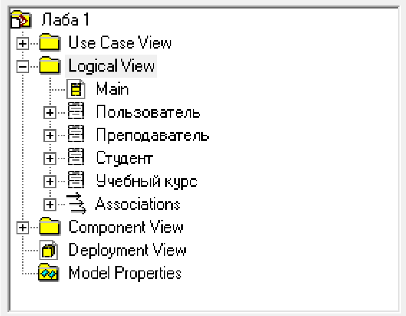


Рисунок 4.9 – Браузер проектов, отображающий все созданные классы

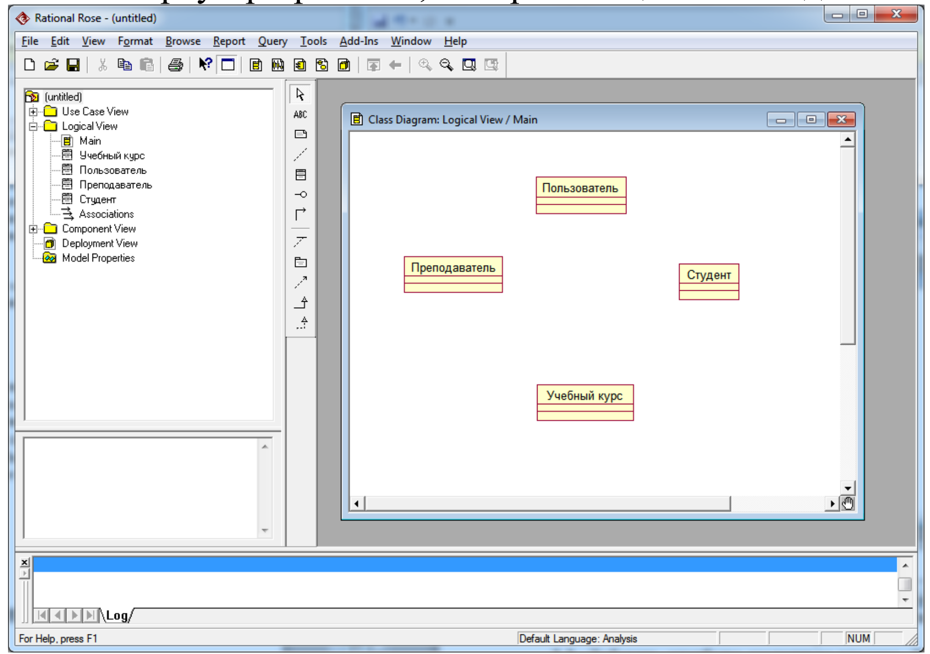


Рисунок 4.10 – Созданные классы в окне диаграммы классов

2.3. Добавить атрибуты созданным классам. Для класса **Пользователь** добавить атрибуты **Имя** и **Ид. номер**; для класса **Преподаватель** – атрибуты **Стаж работы**, **Имя** и **Ид. номер**; для класса **Студент** – атрибуты **Курс**, **Имя** и **Ид. номер**. Для класса **Учебный курс** атрибуты не добавлять.

Добавить атрибут можно одним из следующих способов:

1) В окне диаграммы классов щелкнуть правой кнопкой на графическом изображении нужного класса и выполнить команду контекстного меню **New Attribute**. В этом случае активизируется курсор ввода текста в области графического изображения класса на диаграмме

2) В окне браузера проекта щелкнуть правой кнопкой на нужном классе и выполнить команду контекстного меню **New Attribute**. В этом случае активизируется курсор ввода текста в области иерархического представления класса в браузере проекта под именем соответствующего класса.

3) Дважды щелкнуть на графическом изображении нужного класса. Откроется диалоговое окно свойств **Class Specification** соответствующего класса, в котором выбрать вкладку **Attributes**. В рабочем поле вкладки **Attributes** щелкнуть правой кнопкой мыши и выполнить команду контекстного меню **Insert**.

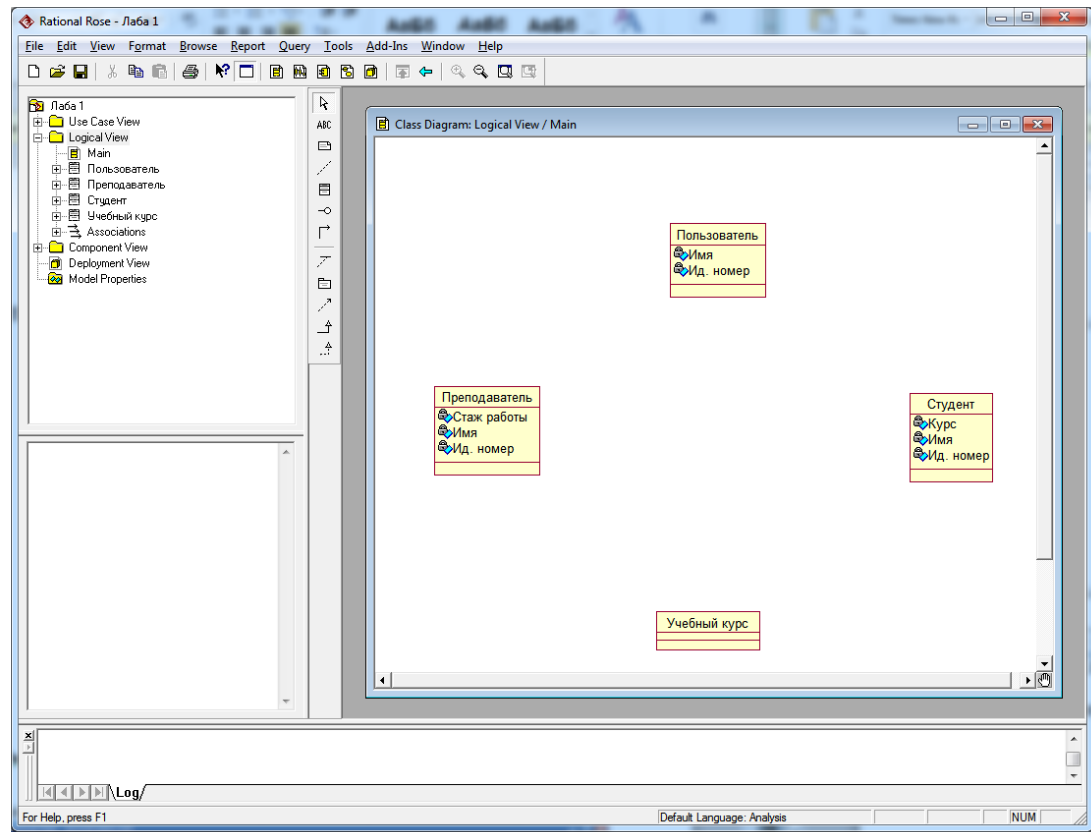


Рисунок 4.11 – Классы с атрибутами в окне диаграммы классов

Контрольные вопросы:

1. Зачем на диаграмме деятельности используется синхронизационная черта?
2. Какое назначение диаграммы компонентов?

Список литературы:

1. Хлебников, А. А. Информационные технологии : учебник / А. А. Хлебников. – М. :КноРус, 2014. – 472 с.
2. Золотов, С.Ю. Проектирование информационных систем : учебное пособие / С.Ю. Золотов ; Министерство образования и науки Российской Федерации, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). - Томск : Эль Контент, 2013. - 88 с. : табл., схем. - ISBN 978-5-4332-0083-8 ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=208706

Лабораторная работа №14. Установка связей между классами

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Существует четыре типа связей в UML:

Зависимость

Ассоциация

Обобщение

Реализация

Эти связи представляют собой базовые строительные блоки для описания отношений в UML, используемые для разработки хорошо согласованных моделей.

Первая из них – зависимость – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит еще одна; может быть снабжена меткой.

Зависимость

Зависимость – это связь использования, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, которые используют ее.

Ассоциация – это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами.

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому.

Например, класс Человек и класс Школа имеют ассоциацию, так как человек может учиться в школе. Ассоциации можно присвоить имя «учится в». В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации.

Ассоциация

Двойные ассоциации представляются линией без стрелок на концах, соединяющей два классовых блока.

Ассоциация может быть именованной, и тогда на концах представляющей её линии будут подписаны роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.

Множественность ассоциации представляет собой диапазон целых чисел, указывающий возможное количество связанных объектов. Он записывается в виде выражения с минимальным и максимальным значением; для их разделения используются две точки. Устанавливая множественность дальнего конца ассоциации, вы указываете, сколько объектов может существовать на дальнем конце ассоциации для каждого объекта класса, находящегося на ближнем ее конце. Количество объектов должно находиться в пределах заданного диапазона. Множественность может быть определена как единица 1, ноль или один 0..1, любое значение 0..* или *, один или несколько 1..*. Можно также задавать диапазон целых значений, например 2..5, или устанавливать точное число, например 3.

Множественность ассоциации

Агрегация – особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое).

Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём, по умолчанию агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

Графически агрегация представляется пустым ромбом на блоке класса «целое», и линией, идущей от этого ромба к классу «часть».

Агрегация

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.

Композиция – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Графически представляется как и агрегация, но с закрашенным ромбиком.

Композиция

Третья связь – обобщение – выражает специализацию или наследование, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет

структуру и поведение родителя. Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя.

Обобщение

Четвертая – реализация – это семантическая связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться. Это связи между интерфейсами и классами, которые реализуют эти интерфейсы. Это, своего рода, отношение «целое-часть». Поставщик, как правило, представлен абстрактным классом. В графическом исполнении связь реализации – это гибрид связей обобщения и зависимости: треугольник указывает на поставщика, а второй конец пунктирной линии – на клиента.

Практическая часть:

2.4. Установить взаимосвязи между классами. Для этого:

2.4.1. С помощью инструмента **Unidirectional Association** специальной панели инструментов построить взаимосвязь типа ассоциация между классами **Преподаватель** и **Учебный курс**. В диалоговом окне спецификации свойств ассоциации **Association Specification** (открывается двойным щелчком мыши на линии взаимосвязи) перейти на вкладку **Role B Detail**, отключить флажок **Navigable**, чтобы убрать направленность отношения, а в раскрывающемся списке **Multiplicity** выбрать кратность ассоциации равную 1. Далее перейти на вкладку **Role A Detail**, отключить флажок **Navigable**, в раскрывающемся списке **Multiplicity** выбрать кратность ассоциации "один ко многим" (1.. n), и вместо n поставить 5. Таким образом, мы указали кратность ассоциации: со стороны **Преподавателя** 1, а со стороны **Учебного курса** – диапазон кратности от 0 до 5. Это означает, что один преподаватель может вести от 0 до 5 учебных курсов.

2.4.2. Аналогично создать взаимосвязь между классами **Студент** и **Учебный курс**, считая, что в группе может быть от 3 до 10 студентов, которые могут быть слушателями от 0 до 4 курсов.

2.4.3. С помощью инструмента **Generalization** специальной панели инструментов построить взаимосвязь типа обобщение между классами **Преподаватель**, **Студент** и **Пользователь**, учитывая, что в данном случае родитель – **Пользователь**.

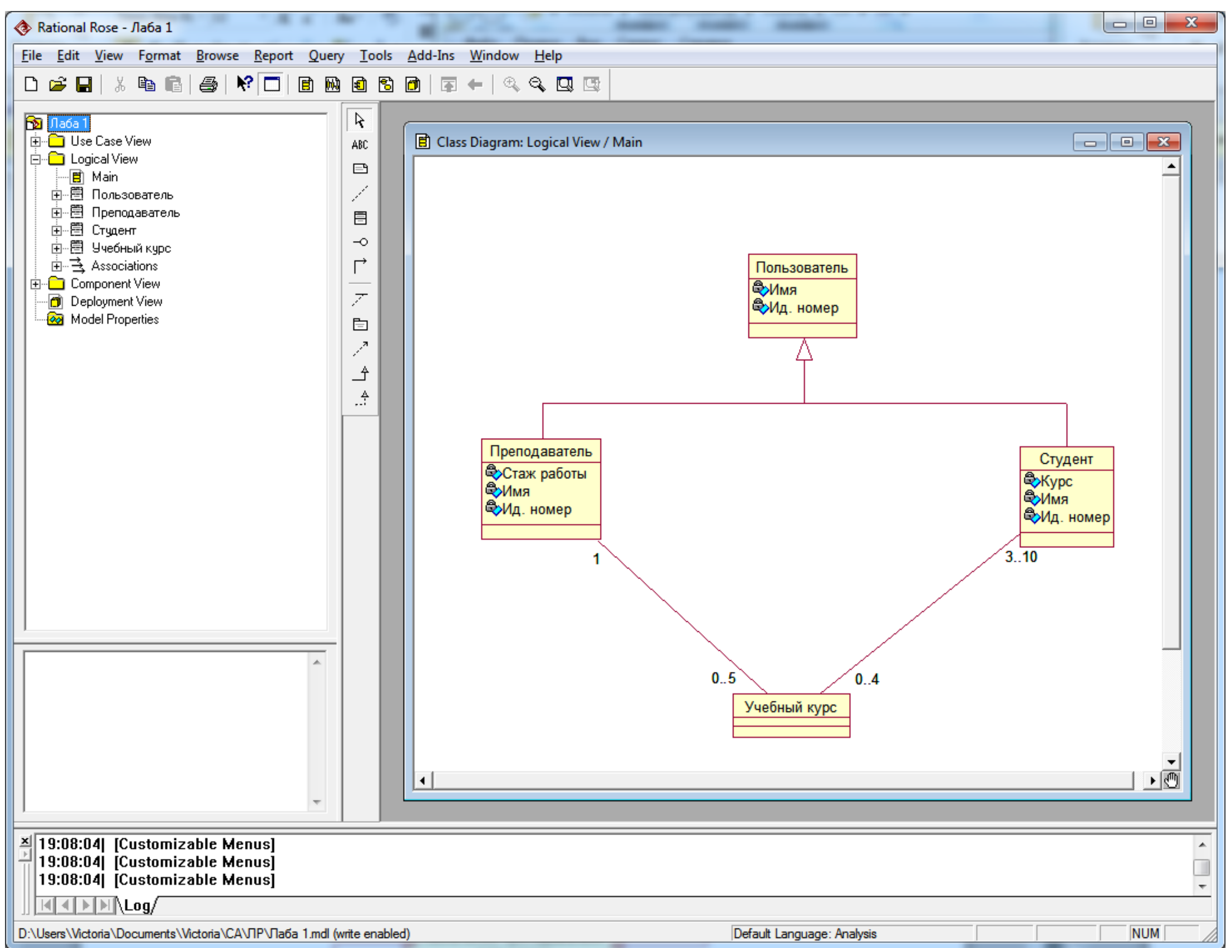


Рисунок 4.12 – Окончательный вариант простой диаграммы классов

Контрольные вопросы:

1. Как создать диаграмму компонентов в программе Rational Rose?
2. Опишите последовательность генерации программного кода в Rational Rose.

Список литературы:

1. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
2. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

Лабораторная работа №15. Моделирование программных систем

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Система Rational Rose – признанный лидер среди средств визуального моделирования, используя ее можно интерактивно разрабатывать архитектуру создаваемого приложения, генерировать его исходные тексты и параллельно работать над документированием разрабатываемой системы.

Преимущества применения Rational Rose:

- сокращение цикла разработки приложения;
- увеличение продуктивности работы программистов;
- улучшение потребительских качеств создаваемых программ за счет ориентации на пользователей и бизнес;
- способность вести большие проекты и группы проектов;
- возможность повторного использования уже созданного ПО за счет упора на разбор их архитектуры и компонентов.

Практическая часть:

Диаграммы, использующиеся в работе

Class diagram (диаграммы классов). Этот вид диаграмм позволяет создавать логическое представление системы, на основе которого создается исходный код описанных классов. Диаграмма классов служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования и содержит детальную информацию об архитектуре программной системы.

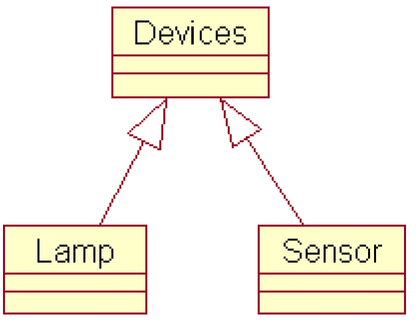


Рисунок 5.1 – Пример диаграммы классов

Activity diagram (диаграммы деятельности или активности). Каждый объект системы, обладающий определенным поведением, может находиться в определенных состояниях, переходить из состояния в состояние, совершая определенные действия в процессе реализации сценария поведения объекта. Поведение большинства объектов реальных систем можно представить с точки зрения теории конечных автоматов, то есть поведение объекта отражается в его состояниях. Для графического отображения состояния объекта используется два вида диаграмм: **Statechart diagram** (диаграмма состояний) и **Activity diagram** (диаграмма активности).

Диаграмма состояний предназначена для отображения состояний объектов системы, имеющих сложную модель поведения.

Диаграмма активности представляет собой дальнейшее развитие диаграммы состояний. Этот тип диаграмм позволяет показать не только последовательность процессов, но и ветвление и даже синхронизацию процессов, позволяет проектировать алгоритмы поведения объектов любой сложности, в том числе может использоваться для составления блок-схем.

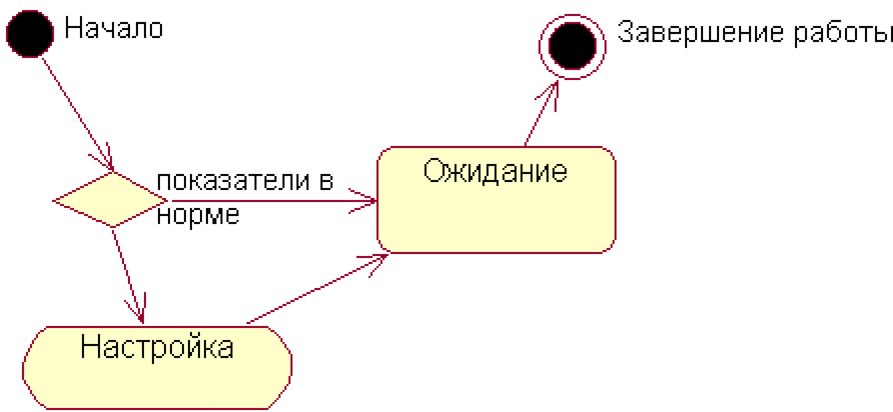


Рисунок 5.2 – Пример диаграммы деятельности

Component diagram (диаграммы компонентов). Этот тип диаграмм предназначен для распределения классов и объектов по компонентам при физическом проектировании системы. Часто данный тип диаграмм называют диаграммами модулей. Диаграмма компонентов служит частью физического представления модели и является необходимой для генерации программного кода.

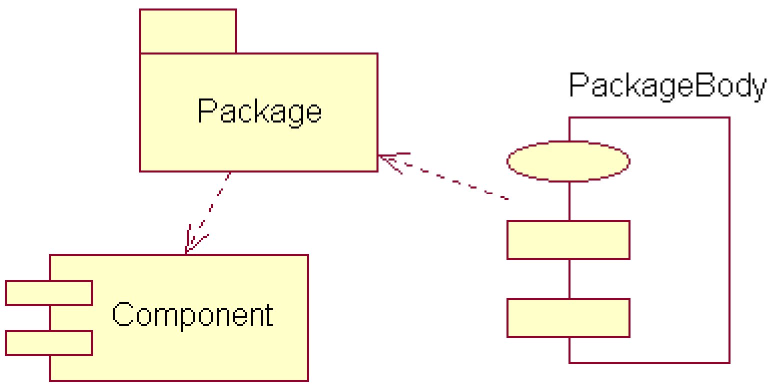


Рисунок 5.3 – Пример диаграммы компонентов

Отношения на диаграммах

Между компонентами диаграмм могут существовать различные отношения, которые описывают их взаимодействие. В языке UML имеется несколько стандартных видов отношений.

Ассоциация – это структурное двунаправленное отношение, описывающее совокупность взаимоотношений между объектами. Ассоциация может иметь имя и кратность. Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Наиболее распространенными являются следующие формы записи кратности отношения ассоциации:

1. Целое неотрицательное число (включая цифру 0). Предназначено для указания кратности, которая является строго фиксированной для элемента соответствующей ассоциации.

2. Два целых неотрицательных числа, разделенные двумя точками и записанные в виде: "первое число .. второе число", например, 1..5. Очевидно, что первое число должно быть строго меньше второго числа в арифметическом смысле, при этом первое число может быть равно 0.

3. Два символа, разделенные двумя точками.При этом первый из них является целым неотрицательным числом или 0, а второй – символом "*" или "n". Здесь символ "*" или "n" обозначает произвольное конечное целое неотрицательное число,значение которого неизвестно на момент задания соответствующего отношения ассоциации.

4. Единственный символ "*" или "n". Является сокращением записи интервала формы записи 3.

Если кратность отношения ассоциации не указана, то по умолчанию принимается ее значение, равное 1.

Частным случаем ассоциации является отношение типа "часть/целое". Отношение такого типа называется **агрегированием**. Агрегирование изображается в виде ассоциации с незакрашенным ромбом со стороны целого.

Обобщение – это однонаправленное отношение, называемое "потомок/прародитель", в котором объект "потомок" может быть подставлен вместо объекта прародителя (родителя или предка). Потомок наследует структуру и поведение своего родителя. Графически обозначается сплошной линией со стрелкой в форме незакрашенного треугольника. Стрелка всегда указывает на родителя.

Контрольные вопросы:

1. Как добавить описание к варианту использования в окне программы Rational Rose?
2. Как прикрепить текстовый файл с описанием к варианту использования?

Список литературы:

1. Михеева, Е. В. Информационные технологии в профессиональной деятельности :учеб.пособие / Е.В. Михеева. - 14-е изд., стер. - М. : Академия, 2016. - 384 с.
2. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.

Лабораторная работа №16. Логическая схема базы данных

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Диаграммой классов в терминологии UML называется диаграмма, на которой показан набор классов (и некоторых других сущностей, не имеющих явного отношения к проектированию БД), а также связеймежду этими классами (иногда термин relationship переводится на русский язык не как “связь”, а как “отношение”). Кроме того, диаграмма классов может включать комментарии и ограничения. Ограничения могут неформально задаваться на естественном языке или же могут формулироваться на языке OCL (Object Constraints Language), который является частью общей спецификации UML, но, в отличие от других частей языка, имеет не графическую, а линейную нотацию. Мы затронем эту тему ниже немного более подробно.

Практическая часть:

1.1. В новом окне самостоятельно создать диаграмму классов, которая состоит из следующих классов:

- Вуз;
- Факультет;
- Студент;
- Курс;
- Преподаватель.

1.2. Для созданных классов установить атрибуты как показано на рис. 5.4.

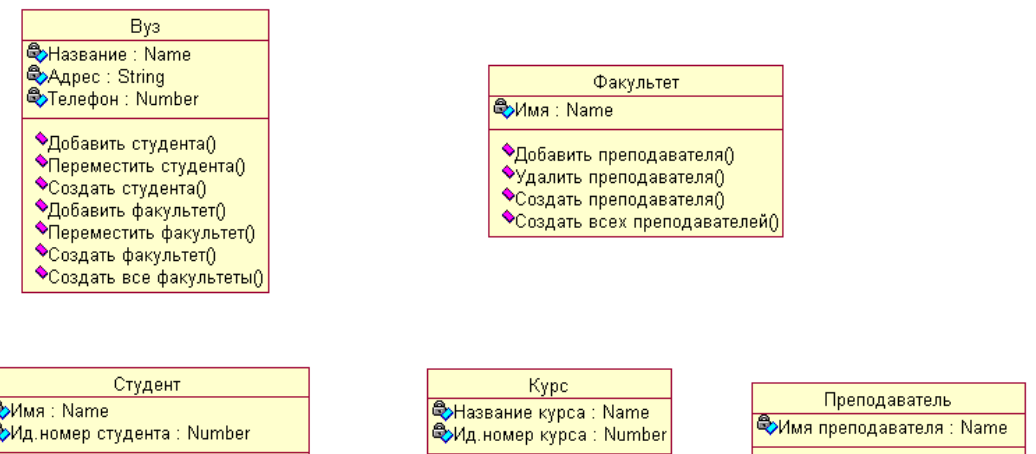


Рисунок 5.4 – Диаграмма классов с атрибутами

1.3. Установить взаимосвязи между классами в соответствии с рис. 5.5. Если кнопка агрегации отсутствует на специальной панели инструментов, ее следует добавить с помощью операции контекстного меню **Customize**.

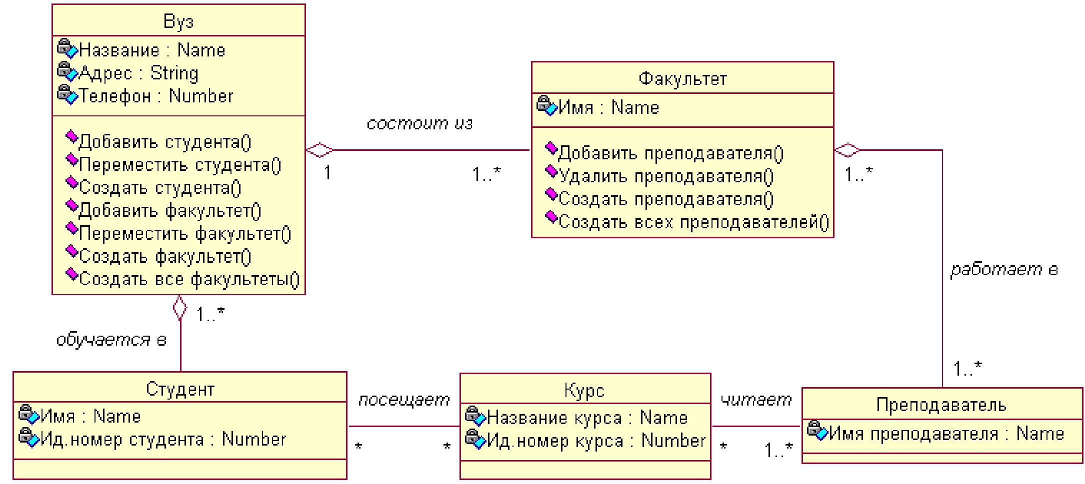


Рисунок 5.5 – Диаграмма классов, моделирующая объекты системы регистрации курсов и отношения между ними

Контрольные вопросы:

1. Что представляет собой реализация конкретного варианта использования?
2. Как создать элемент Кооперация в окне браузера проекта?

Список литературы:

1. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
2. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

Лабораторная работа №17. Диаграмма деятельности

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Диагра́мма де́ятельности (англ. activity diagram) — UML-диаграмма, на которой показаны действия, состояния которых описано на диаграмме состояний. Под деятельностью (англ. activity) понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий англ. action, соединённых между собой потоками, которые идут от выходов одного узла ко входам другого.

Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений.

Диаграммы деятельности состоят из ограниченного количества фигур, соединённых стрелками. Основные фигуры:

Прямоугольники с закруглениями — действия

Ромбы — решения

Широкие полосы — начало (разветвление) и окончание (схождение) ветвления действий

Чёрный круг — начало процесса (начальный узел)

Чёрный круг с обводкой — окончание процесса (финальный узел)

Стрелки идут от начала к концу процесса и показывают потоки управления или потоки объектов (данных).

Практическая часть:

Открыть рабочее окно для построения диаграммы деятельности **Activity Diagram** с помощью контекстного меню представления **Use Case View** в браузере проекта.

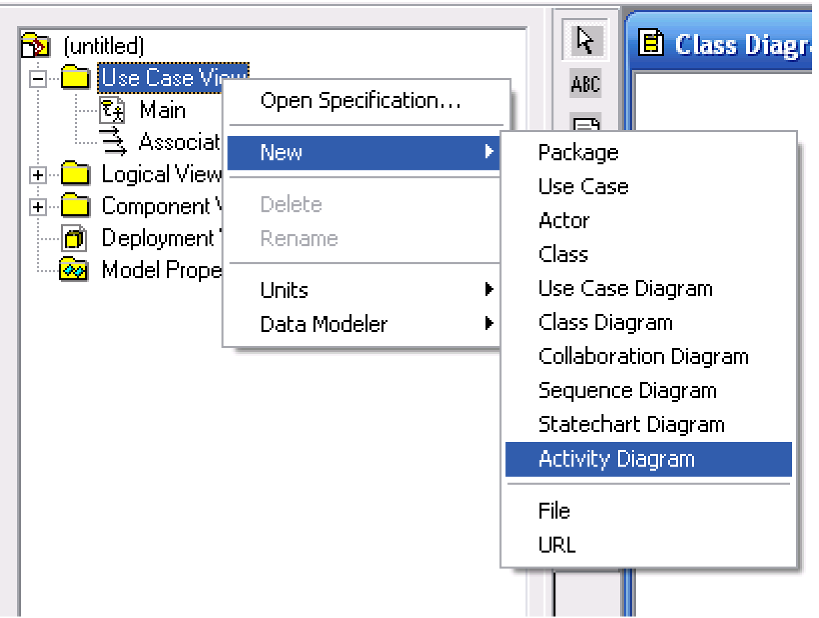


Рисунок 5.6 – Вид контекстного меню создания диаграммы деятельности

В результате появится новое окно с чистым рабочим листом диаграммы деятельности, а на специальной панели инструментов – кнопки, необходимые для разработки диаграммы деятельности . Назначение отдельных кнопок панели можно узнать из всплывающих подсказок.

2.2. Создать деятельности в окне браузера проекта в соответствии с рисунком 5.7.

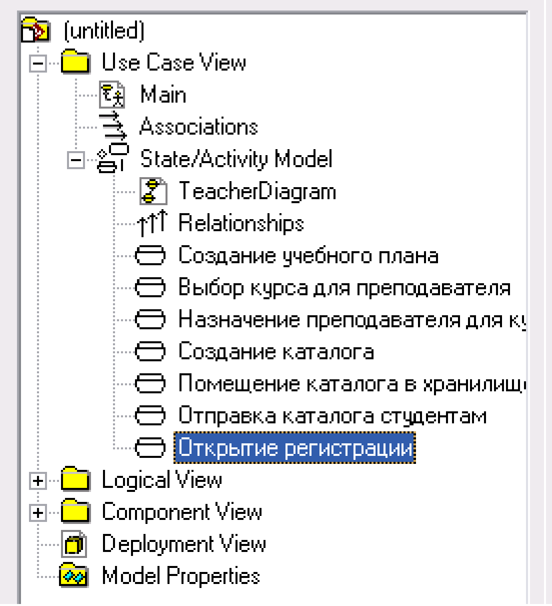


Рисунок 5.7 – Окно браузера проекта

2.3. Созданные деятельности перетащить в окно диаграммы деятельности и расположить как на рисунке 5.8.

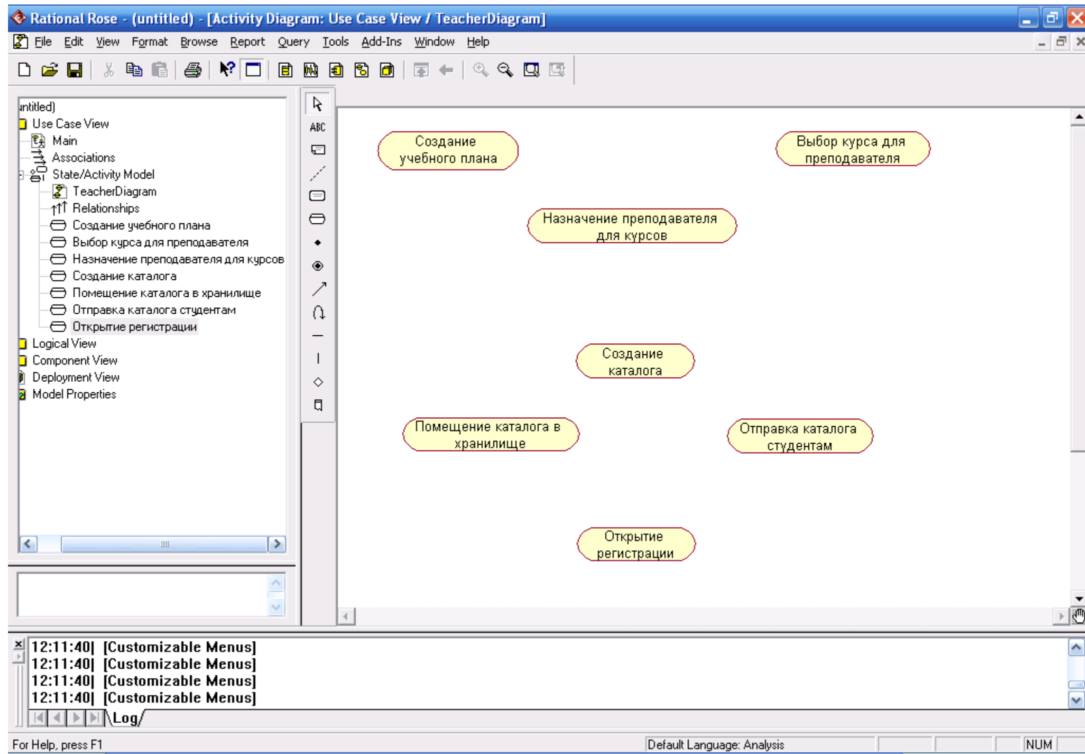


Рисунок 5.8 – Окно диаграммы деятельности

2.4. С помощью специальной панели инструментов добавить на диаграмму элемент принятия решения (ветвления) для альтернативных переходов ◇ и расположить его как на рис. 5.9.

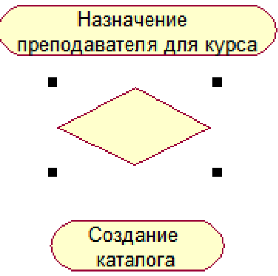


Рисунок 5.9 – Фрагмент окна диаграммы деятельности

2.5. С помощью команды **Open Specification …** контекстного меню элемента ветвления задать вопрос условия перехода: "Все ли преподаватели назначены?" (рис. 5.10).

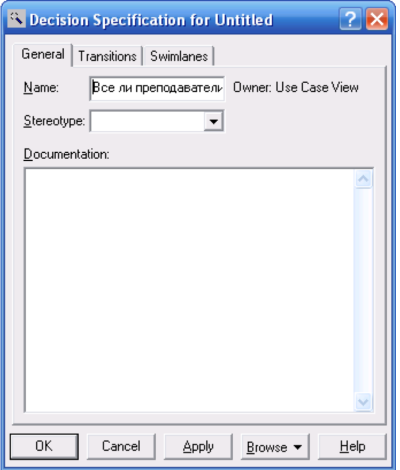


Рисунок 5.10 – Диалоговое окно свойств объекта ветвления диаграммы

2.6. С помощью специальной панели инструментов добавить на диаграмму синхронизационную черту для отображения условия, соответствующего логическому оператору "и", а также переходы между деятельностями как на рис. 5.11.

Рисунок 5.11 – Диаграмма деятельности, которая моделирует действия, выполняемые в процессе создания системы регистрации учебных курсов

Контрольные вопросы:

1. Что представляет собой элемент Кооперация?
2. В чем заключается идентификация ключевых абстракций?
3. Что такое граничные классы?

Список литературы:

1. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия,

2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.

2. Хлебников, А. А. Информационные технологии : учебник / А. А. Хлебников. – М. :КноРус, 2014. – 472 с.

Лабораторная работа №18. Диаграмма классов

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Диаграмма классов (англ. Static Structure diagram) — структурная диаграмма языка моделирования UML, демонстрирующая общую структуру иерархии классов системы, их коопераций, атрибутов (полей), методов, интерфейсов и взаимосвязей между ними. Широко применяется не только для документирования и визуализации, но также для конструирования посредством прямого или обратного проектирования.

Практическая часть:

1. Figure (фигура);
2. Figure Element (элемент фигуры);
3. Point (точка);
4. Line (линия).

Порядок выполнения:

3.1. Создать и расположить все описанные выше классы в соответствии с рис. 5.12.

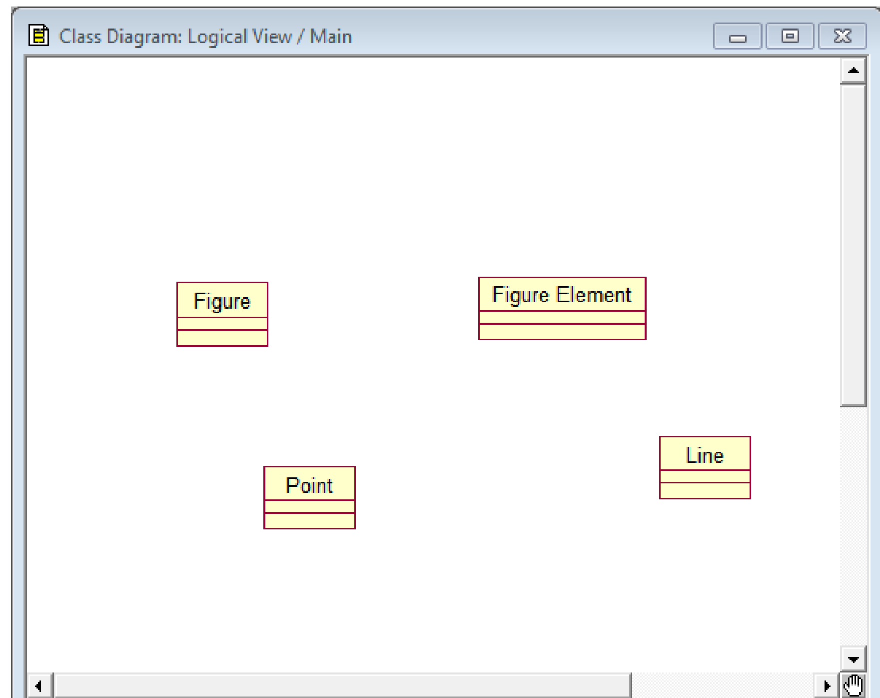


Рисунок 5.12 – Окно диаграммы классов графического редактора

3.2. Созданной диаграмме дать имя **Display**. Для этого воспользоваться инструментом **TextBox** ABC на специальной панели инструментов.

3.3. Изменить стиль форматирования надписи с помощью команд **Format...** → **Font...** из контекстного меню надписи. Установить шрифт полужирный, подчеркнутый, размер – 12 пт.

3.4. Для класса **Figure** добавить операции как на рис. 5.13.

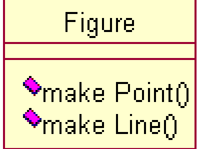


Рисунок 5.13 – Операции класса **Figure**

3.5. Для класса **Figure Element** добавить операцию с аргументами **move By(int , int)** . Операция **move By** перемещает объект в точку экрана с координатами (**X : integer; Y : integer**). Чтобы задать аргументы операции необходимо в диалоговом окне **Class Specification for Figure Elemen** щелкнуть правой кнопкой на имени операции и в контекстном меню выбрать пункт **Specification…**

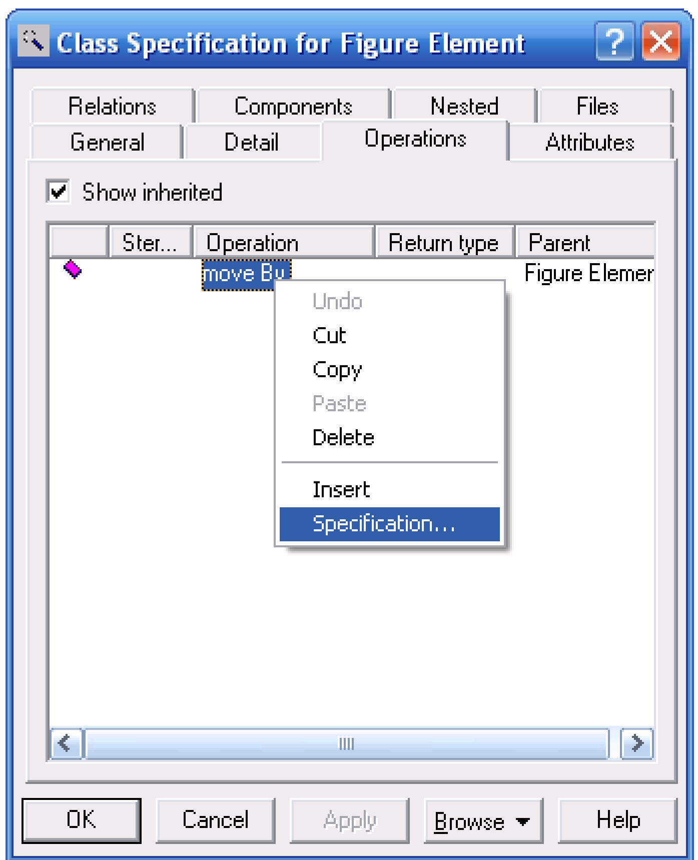


Рисунок 5.14 – Диалоговое окно для задания дополнительных свойств операции **move By**

В этом диалоговом окне на вкладке **Detail** с помощью команды **Insert** контекстного меню задать аргументы **X** типа **Integer** и **Y** типа **Integer**.

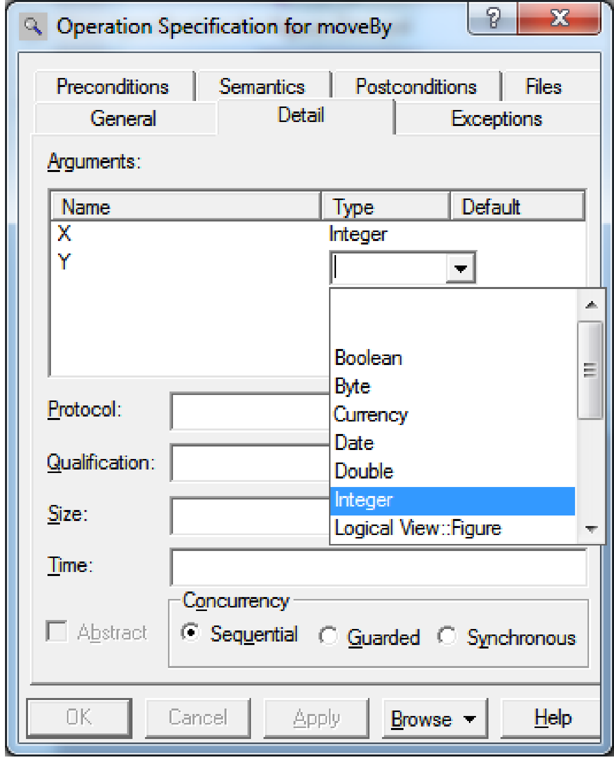


Рисунок 5.15 – Диалоговое окно аргументов операции **move By**

Если все сделано правильно, то после щелчка на операции **move By** в окне диаграммы классов появятся аргументы этой операции как на рис. 5.16.

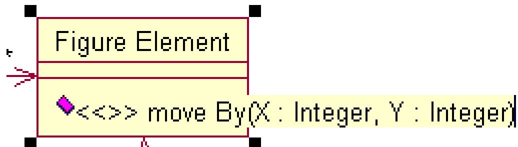


Рисунок 5.16 – Отображение аргументов операции **move By** класса **Figure Element**

3.6. Для класса **Point** задать следующие операции:

- get X();
- get Y();
- set X(int);
- set Y(int);
- move By(int;int).

3.7. Для класса **Line** задать следующие операции:

- get P1();
- get P2();
- set P1(Point);
- set P2(Point);
- move By(int;int).

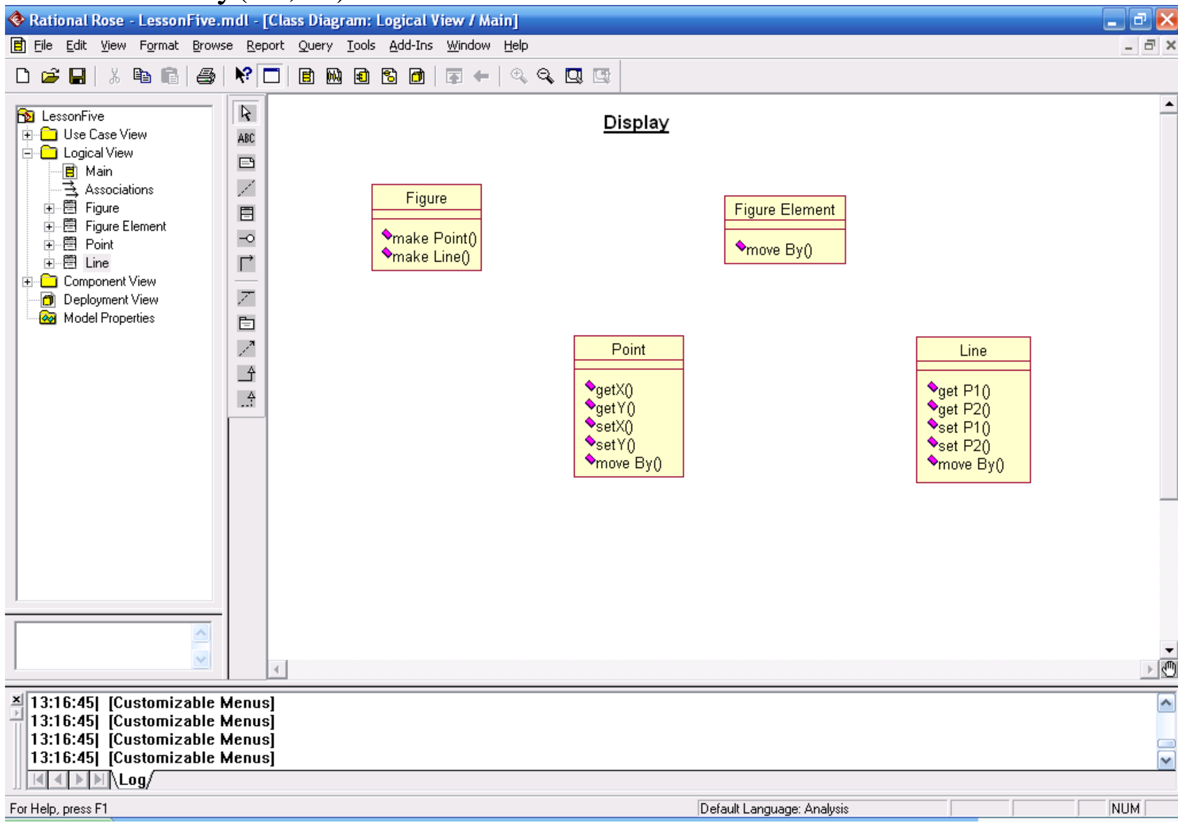


Рисунок 5.17 – Окно диаграммы классов с заданными отношениями

3.8. Установить взаимосвязи между классами как на рис. 5.18.

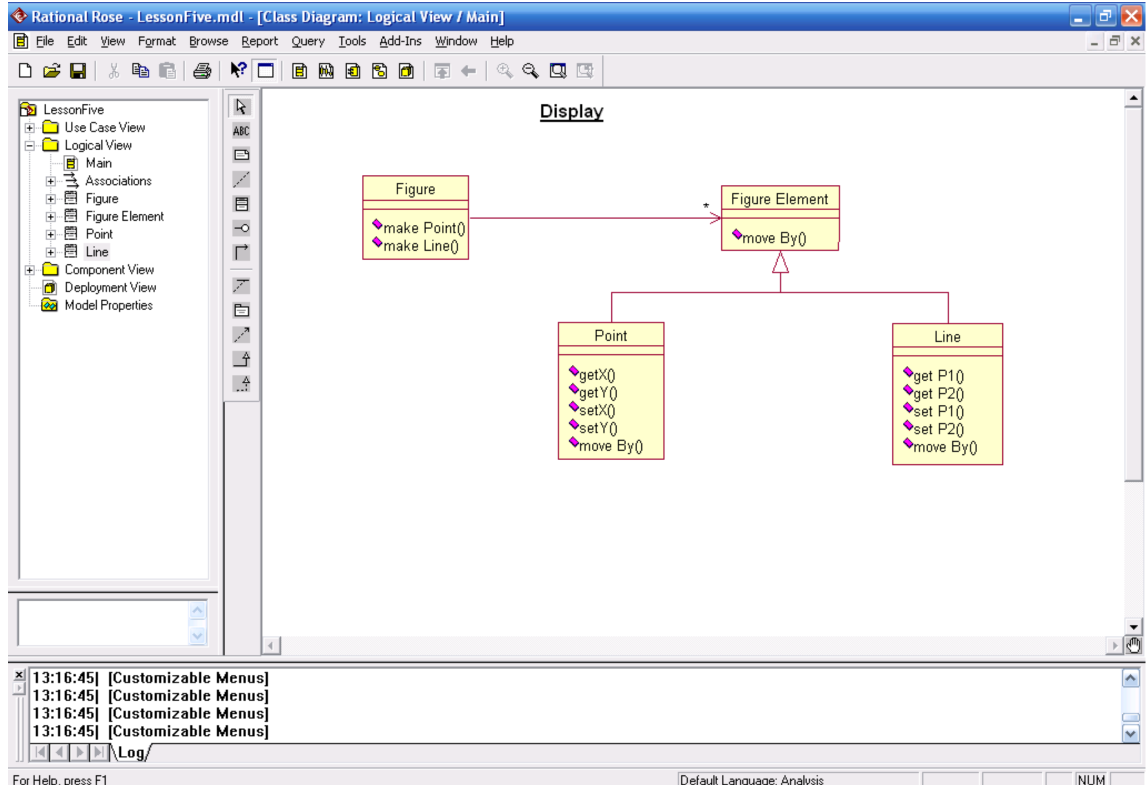


Рисунок 5.18 – Диаграмма классов графического редактора

Контрольные вопросы:

1. Как создать граничный класс в программе Rational Rose?
2. Что представляют собой классы-сущности?

Список литературы:

1. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.
2. Хлебников, А. А. Информационные технологии : учебник / А. А. Хлебников. – М. :КноРус, 2014. – 472 с.

Лабораторная работа №19. Генерация программного кода

Цель работы:

Приобрести навыки моделирования баз данных на примере построения диаграмм классов и деятельности, а также генерации программного кода на примере проектирования простого графического редактора.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Общая последовательность действий, которые необходимо выполнить для генерации программного кода в среде IBM Rational Rose 2003, состоит из следующих этапов:

Проверка модели на отсутствие ошибок.

Создание компонентов для реализации классов.

Отображение классов на компоненты.

Выбор языка программирования для генерации текста программного кода.

Установка свойств генерации программного кода.

Выбор класса, компонента или пакета.

Генерация программного кода.

Особенности выполнения каждого из этапов могут изменяться в зависимости от выбора языка программирования или схемы базы данных.

Практическая часть:

4.1. Проверить модель на отсутствие ошибок. Для этого выполнить операцию главного меню: **Tools → Check Model** (Инструменты → Проверить модель). Результаты проверки разработанной модели на наличие ошибок отображаются в окне журнала. Прежде чем приступить к генерации текста программного кода разработчику следует добиться устранения всех ошибок и предупреждений, о чем должно свидетельствовать чистое окно журнала (рис. 5.19).

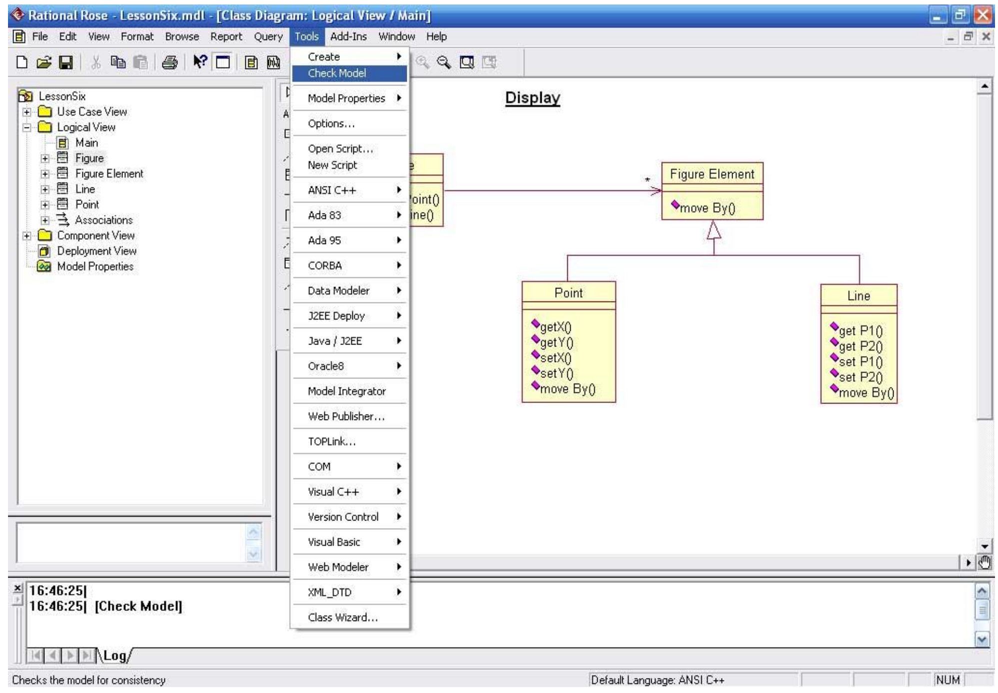


Рисунок 5.19 – Проверка модели на отсутствие ошибок

4.2. Создать компоненты для реализации классов. В браузере проекта предназначено отдельное представление компонентов **Component View**, в котором уже содержится диаграмма компонентов с пустым содержанием и именем по умолчанию **Main** (Главная). Для активизации диаграммы компонентов в браузере проекта раскрыть представление **Component View** и дважды щелкнуть на пиктограмме **Main**. В результате выполнения этих действий появляется новое окно с чистым рабочим листом диаграммы компонентов и специальная панель инструментов, содержащая кнопки для разработки диаграммы компонентов. Далее добавить в область диаграммы компонентов элемент **Component** и задать ему имя **MainPaint.exe**.

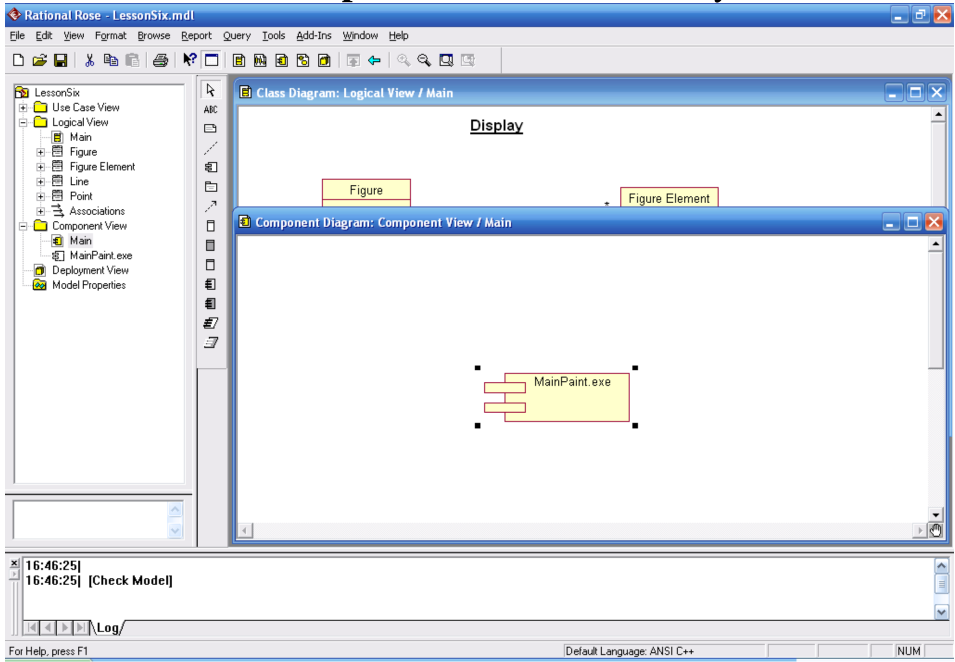


Рисунок 5.20 – Диаграмма компонентов

4.3. Отобразить классы на компоненты. Для этого воспользоваться окном спецификации свойств компонента, открытого на вкладке **Realizes** (Реализует). Для включения реализации класса в данный компонент следует выделить требуемый класс на этой вкладке и выполнить для него операцию контекстного меню **Assign** (Назначить). В результате перед именем класса на этой вкладке появится специальная отметка.

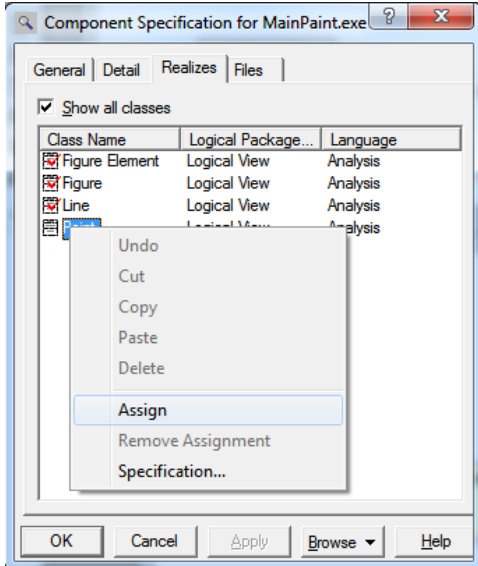


Рисунок 5.21 – Связывание классов с компонентом

4.4 . Выбрать языка программирования для генерации программного кода.

Для выбора языка **ANSI C++** в качестве языка реализации модели следует выполнить операцию главного меню: **Tools** → **Options** (Инструменты → Параметры), в результате чего будет вызвано диалоговое окно настройки параметров модели. Далее на вкладке **Notation** (Нотация) в строке **Default Language** (Язык по умолчанию) из вложенного списка следует выбрать язык **ANSI C++**.

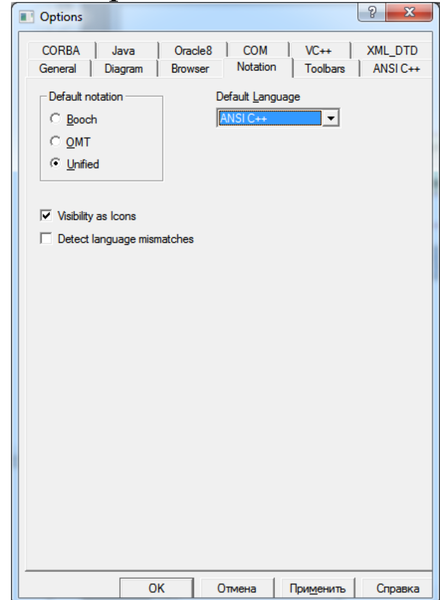


Рисунок 5.22 – Диалоговое окно выбора языка программирования

После выбора языка программирования по умолчанию следует изменить язык реализации каждого из компонентов модели. Для этого в окне диаграммы компонентов с помощью контекстного меню компонента открыть окно спецификации свойств компонента и на вкладке **General** (Общие) в строке **Language** (Язык) из вложенного списка выбрать язык **ANSI C++**.

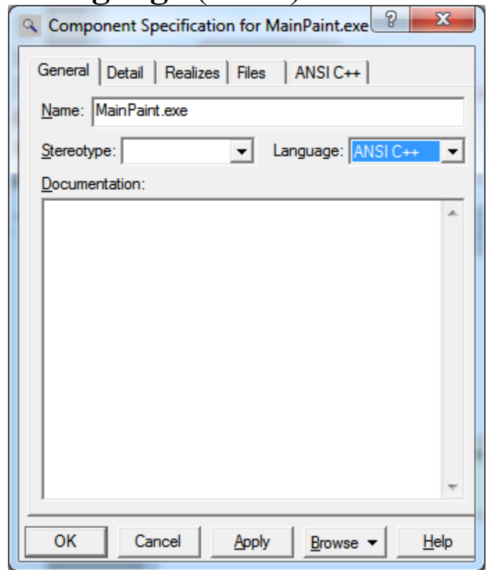


Рисунок 5.23 – Диалоговое окно выбора языка реализации компонентов

4.5. Сгенерировать программный код.

Генерация программного кода в среде IBM Rational Rose возможна для отдельного класса или компонента . Для этого выполнить операцию контекстного меню компонента **ANSI C++ → Generate Code…** (Язык ANSI C++ → Генерировать код). В результате этого будет открыто диалоговое окно, в котором необходимо указать путь для сохранения сгенерированных файлов, а затем окно с предложением выбора классов для генерации программного кода на выбранном языке программирования.

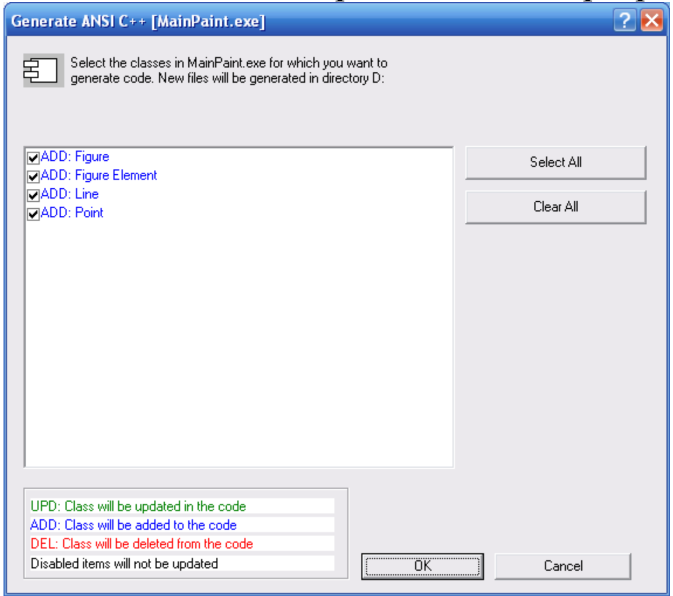


Рисунок 5.24 – Окно выбора классов для генерации программного кода

После генерации программного кода для компонента **MainPaint.exe** каждому классу, реализованному в данном компоненте, будет соответствовать 2 файла с текстом кода на языке **ANSI C++**:

- заголовочный файл с расширением **.h**;
- файл реализации с расширением **.cpp**.

Figure Element.cpp
Figure Element.h
Figure.cpp
Figure.h
Line.cpp
Line.h
Point.cpp
Point.h

Рисунок 5.25 – Сгенерированные файлы

Контрольные вопросы:

1. Какие файлы создаются при генерации программного кода?
2. Какие элементы содержит специальная панель инструментов для создания диаграммы компонентов?

Список литературы:

3. Гохберг, Г. С. Информационные технологии : учебник / Г.С. Гохберг, А.В. Зафиевский, А.А. Короткин. - 9-е изд., перераб. и доп. - М. : Академия, 2014. - 240 с.
4. Белов, В. В. Проектирование информационных систем : учебник / В.В. Белов, В.И. Чистяков ; под ред. В.В. Белова. - М. : Академия, 2013. - 352 с. - (Бакалавриат). - На учебнике гриф: Рек.УМО. - Библиогр.: с. 345-347. - ISBN 978-5-7695-7406-1.