Федеральное государственное автономное образовательное
учреждение высшего образования
«Санкт-Петербургский государственный университет
аэрокосмического приборостроения»

Основы программной инженерии

Методические указания к выполнению лабораторных работ

Составители:

к.т.н., стар. преп. П.А. Охтилев, асс. А.Э. Зянчурин

Рецензент:

д.т.н., проф. М.Ю. Охтилев

Санкт-Петербург – 2021

Содержание

Требования к оформлению отчетов

Структура и форма отчета о лабораторной работе

Итоговый отчёт по лабораторным работам должен состоять из частей, перечисленных ниже. Отсутствие указанных ниже частей не допускается. Общий объем отчёта должен составлять не менее 15 страниц. Страницы должны быть пронумерованы. Размер шрифта основного текста — 14 пунктов.

1. *Титульный лист* должен соответствовать образцу на сайте ГУАП. При оформлении титульного листа обязательно наличие следующей информации:
   - название дисциплины;
   - ФИО преподавателя, принимающего работу;
   - ФИО обучающихся, выполнивших работу.
   - Отчёты, содержащие неверную информацию на титульном листе, к сдаче не принимаются.
2. *Содержание* с указанием номеров страниц (желательно составленное автоматически).
3. *Подписанное преподавателем задание* на лабораторные работы.
4. *Краткое описание хода выполнения работ*: постановка задачи на каждую работу; описание использованных моделей, алгоритмов, программных компонент. Описание должно быть сопровождено расчетными материалами, снимками с экрана компьютера («скриншотами»), отражающими ход выполнения работы.
5. *Выводы* по результатам выполняемых лабораторных работ.
6. *Список цитируемой и использованной литературы.*

Требования к оформлению отчета о лабораторной работе

Изложение текста и оформление лабораторных работ следует выполнять в соответствии с ГОСТ 2.105-2019 – ЕСКД и общие требования к текстовым документам ГОСТ 7.32 – 2017 – СИБИД, соблюдая следующие требования.

1. *Оформление титульного листа.* Титульный лист следует оформлять на бланке. Бланки для оформления титульных листов учебных работ представлены на сайте ГУАП в разделе «Нормативная документация» для учебного процесса.
2. *Оформление основного текста работы:*
   - следует использовать шрифт Times New Roman размером не менее 12 пт (допускается 14 пт), строчный, без выделения, с выравниванием по ширине;
   - абзацный отступ должен быть одинаковым и равен по всему тексту 1,25 см;
   - строки разделяются полуторным интервалом;

- поля страницы: верхнее и нижнее – 20 мм, левое – 30 мм, правое – 15 мм;
- полужирный шрифт применяется только для заголовков разделов и подразделов. заголовков структурных элементов;
- разрешается использовать компьютерные возможности акцентирования внимания на определенных терминах, формулах, теоремах, применяя шрифты разной гарнитуры;
- наименования структурных элементов работы: «СОДЕРЖАНИЕ», «ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ», «ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ», «ВВЕДЕНИЕ», «ЗАКЛЮЧЕНИЕ», «СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ», «ПРИЛОЖЕНИЕ» следует располагать в середине строки без точки в конце, прописными (заглавными) буквами, не подчеркивая;
- введение и заключение не нумеруются.
- каждый структурный элемент и каждый раздел основной части следует начинать с новой страницы.

3. *Оформление основной части работы следует делить на разделы и подразделы:*
   - разделы и подразделы должны иметь порядковую нумерацию в пределах всего текста, за исключением приложений;
   - нумеровать их следует арабскими цифрами;
   - номер подраздела должен включать номер раздела и порядковый номер подраздела, разделенные точкой;
   - после номера раздела и подраздела в тексте точка не ставится;
   - разделы и подразделы должны иметь заголовки;
   - если заголовок раздела, подраздела или пункта занимает не одну строку, то каждая следующая строка должна начинаться с начала строки, без абзацного отступа;
   - заголовки разделов и подразделов следует печатать с абзацного отступа с прописной буквы, полужирным шрифтом, без точки в конце, не подчеркивая;
   - если заголовок состоит из двух предложений, их разделяют точкой;
   - переносы слов в заголовках не допускаются;
   - обозначение подразделов следует располагать после абзацного отступа, равного двум знакам относительно обозначения разделов;
   - обозначение пунктов приводят после абзацного отступа, равного четырем знакам относительно обозначения разделов;
   - в содержании должны приводиться наименования структурных элементов, после заголовка каждого из них ставят отточие и приводят номер страницы;
   - содержание должно включать введение, наименование всех разделов и подразделов, заключение, список использованных источников и наименование приложений с указанием номеров страниц, с которых начинаются эти элементы работы;

- перечень сокращений и обозначений следует располагать в алфавитном порядке. Если условных обозначений в отчете менее трех, перечень не составляется.

4. *Оформление нумерации страниц:*
   - страницы следует нумеровать арабскими цифрами, соблюдая сквозную нумерацию по всему тексту работы;
   - номер страницы следует проставлять в центре нижней части листа без точки;
   - титульный лист должен включаться в общую нумерацию страниц;
   - номер страницы на титульном листе не проставляется.
5. *Оформление рисунков:*
   - на все рисунки должны быть ссылки: …в соответствии с рисунком 1;
   - рисунки, за исключением рисунков приложений, следует нумеровать арабскими цифрами;
   - рисунки могут иметь наименование и пояснительные данные, которые помещаются в строке над названием рисунка: Рисунок 1 – Детали прибора
   - рисунки каждого приложения должны обозначаться отдельной нумерацией арабскими цифрами с добавлением перед цифрой обозначения приложения: Рисунок А.3 (третий рисунок приложения А).
6. *Оформление таблиц:*
   - на все таблицы должны быть ссылки, при ссылке следует писать слово «таблица» с указанием ее номера;
   - таблицы, за исключением таблиц приложений, следует нумеровать арабскими цифрами сквозной нумерацией;
   - наименование таблицы следует помещать над таблицей слева, без абзацного отступа: Таблица 1 – Детали прибора
   - таблицы каждого приложения обозначают отдельной нумерацией арабскими цифрами с добавлением перед цифрой обозначения приложения:
     Таблица Б.2 (вторая таблица приложения Б)
   - если таблица переносится на следующую страницу, под заголовком граф должна быть строка с номером колонок, на следующей странице под названием «Продолжение таблицы 1» дается строка с номером колонок.
7. *Оформление приложений:*
   - в тексте отчета на все приложения должны быть ссылки, приложения располагаются в порядке ссылок на них в тексте отчета;
   - каждое приложение следует размещать с новой страницы с указанием в верхней части страницы слова «ПРИЛОЖЕНИЕ»;
   - заголовок приложения записывают с прописной буквы, полужирным шрифтом, отдельной строкой по центру без точки в конце;

- приложения обозначаются прописными буквами кириллического алфавита, начиная с А, за исключением букв Ё, З, Й О, Ч, Ъ, Ы, Ь;
- допускается обозначение приложений буквами латинского алфавита, за исключением I и O;
- в случае полного использования букв кириллического и латинского алфавита допускается обозначать приложения арабскими цифрами;
- приложение следует располагать после списка использованных источников.

8. *Список использованных источников.* Сведения об источниках следует располагать в порядке появления ссылок на источник и в тексте работы и нумеровать арабскими цифрами с точкой и печатать с абзацного отступа. Список использованных источников следует оформлять в соответствии с ГОСТ 7.0.100 – 2018 «Библиографическая запись. Библиографическое описание». Примеры библиографического описания в соответствии с требованиями ГОСТ 7.0.100 – 2018 представлены на сайте ГУАП в разделе «Нормативная документация» для учебного процесса.

Лабораторная работа №2. Организация технологического процесса промышленного производства программного обеспечения. Применение и конфигурирование общего и системного программного обеспечения. Разработка специального программного обеспечения

Цель работы

Целью работы является формирование практических навыков организации технологического процесса промышленного производства программного обеспечения.

Задание на лабораторную работу

Этап I

Организация технологического процесса промышленного производства программного обеспечения. Установка и настройка виртуальной машины для разработки. Развертывание, конфигурирование и применение систем управления проектами и контроля версий. Постановка задач и распределение зон ответственности в проекте. Выполнение первого этапа данной лабораторной работы предполагает коллективное взаимодействие в группе из трех человек в ролях: «Руководитель проектов в области информационных технологий», «Специалист по информационным системам», «DevOps-инженер».

Этап II

Разработка программного обеспечения в соответствии с требованиями, предъявленными в первой лабораторной работе и поставленными задачами на первом этапе данной лабораторной работы. Выполнение второго этапа данной лабораторной работы предполагает коллективное взаимодействие в группе из трех человек в ролях: «Системный программист», «Программист», «Администратор баз данных».

Общие рекомендации по выполнению лабораторной работы. Этап I

В первом этапе лабораторной работы рассмотрены основные подготовительные мероприятия организации технологического процесса промышленного производства программного обеспечения.

Среды виртуализации и виртуальные машины

Установка и настройка виртуальной машины
Цели
1. Изучить состав инструментальных средств создания и сопровождения аппаратных конфигураций виртуальных машин. 2. Получить практические навыки выполнения типовых операций мониторинга и управления состояниями виртуальных машин. 3. Изучить функциональные возможности интеграции виртуальных (гостевых) и физической (хостовой) машин.

Раздел №1. Введение.

VirtualBox – это свободно распространяемый (по лицензии GNU GPL) программный продукт от компании Oracle, обеспечивающий виртуализацию персональных компьютеров x86/x64 – совместимых архитектур с возможностью установки на них различных операционных систем, а также организации их независимого или совместного (в составе виртуальной компьютерной сети) функционирования.

В настоящих методических указаниях будут рассмотрены минимально необходимые практики работы пользователя в инструментальной среде Oracle VirtualBox.

Раздел №2. Установка VirtualBox.

Менеджер виртуальных машин VirtualBox устанавливается в конфигурацию операционной системы по тем же правилам, что большинство других прикладных программ. В частности, для установки в ОС MS Windows необходимо загрузить установочный файл из официального сайта (https://www.virtualbox.org/wiki/Download_Old_Builds_6_0) и запустить его на исполнение. В результате установки в главном меню ОС и/или на рабочем столе появится ярлык на VirtualBox.

В результате установки VirtualBox будет осуществлен запуск и отображено окно программы.

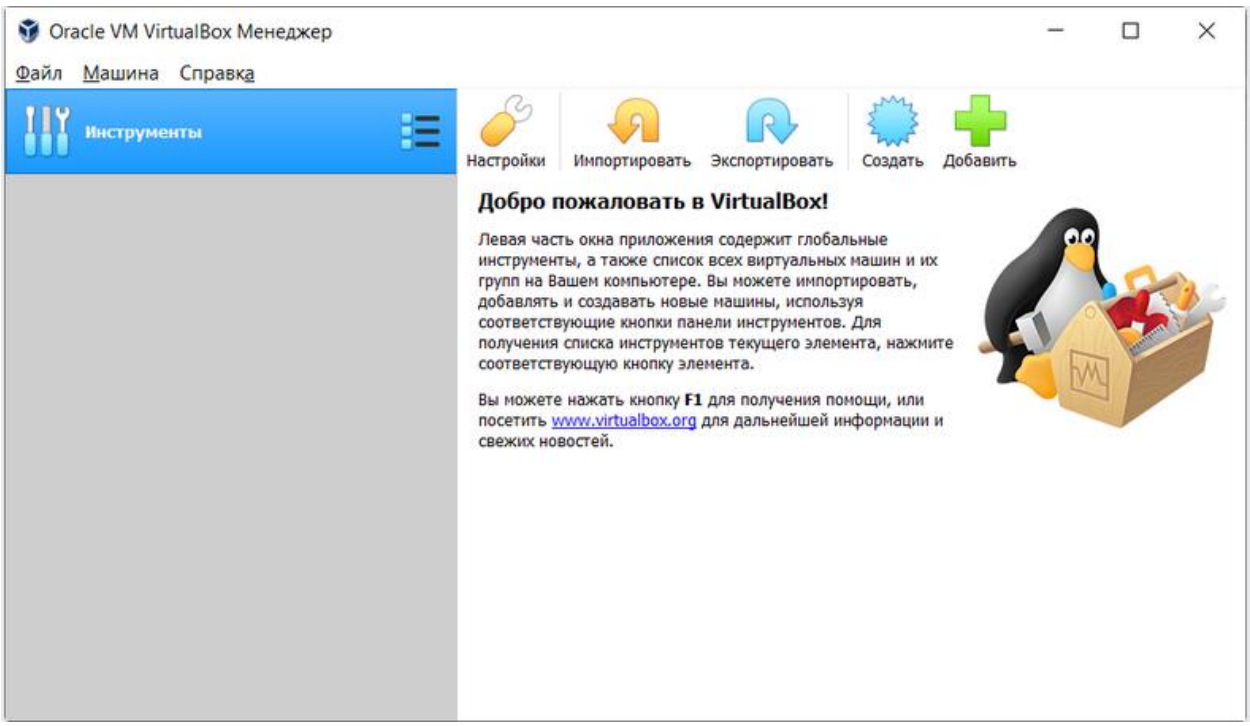


Рис. 1. Начальное окно VirtualBox

Раздел №3. Создание виртуальной машины

Одной из базовых операций в среде VirtualBox является создание новой виртуальной (называемой также гостевой) машины.

Для этого необходимо в главном окне VirtualBox нажать кнопку Создать, активация которой вызовет мастер создания гостевой машины.

На первом шаге необходимо задать имя виртуальной машины и выбрать тип гостевой операционной системы из выпадающего списка. На рисунке приведён пример выбора типа ОС «Linux» и подтипа (внутри типа) «Ubuntu (32 bit)» (Debian (64 bit)). В. Выбор типа и подтипа гостевой ОС из списка обеспечит включение некоторых видов оптимизации для эмуляции работы конкретной ОС.

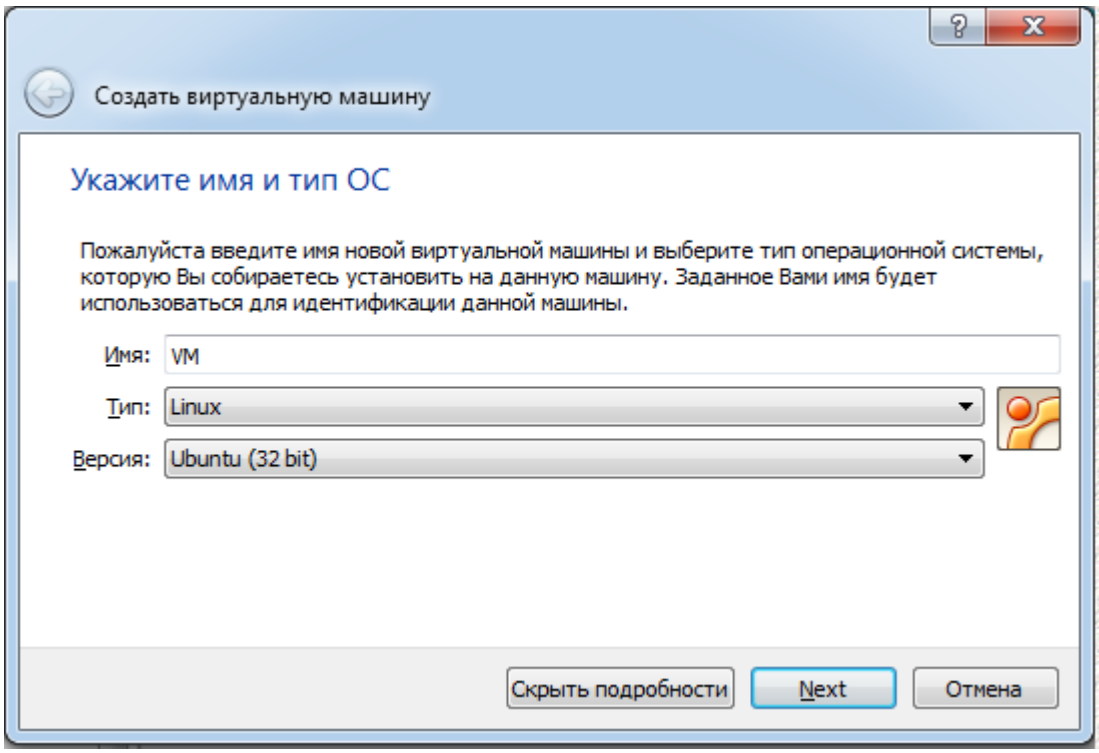


Рис. 2. Создание виртуальной машины

Заметим, что при использовании 64-х битной ОС, в ней можно запускать как 32-х битные, так и 64-х битные программы. На следующем шаге мастера необходимо определить объём оперативной памяти виртуальной машины. Это очень важный параметр виртуальной машины, который существенно влияет на её производительность. Поэтому желательно при наличии достаточного объёма физической памяти в хостовой машине выбирать объём оперативной памяти гостевой машины не меньше рекомендуемого мастером.

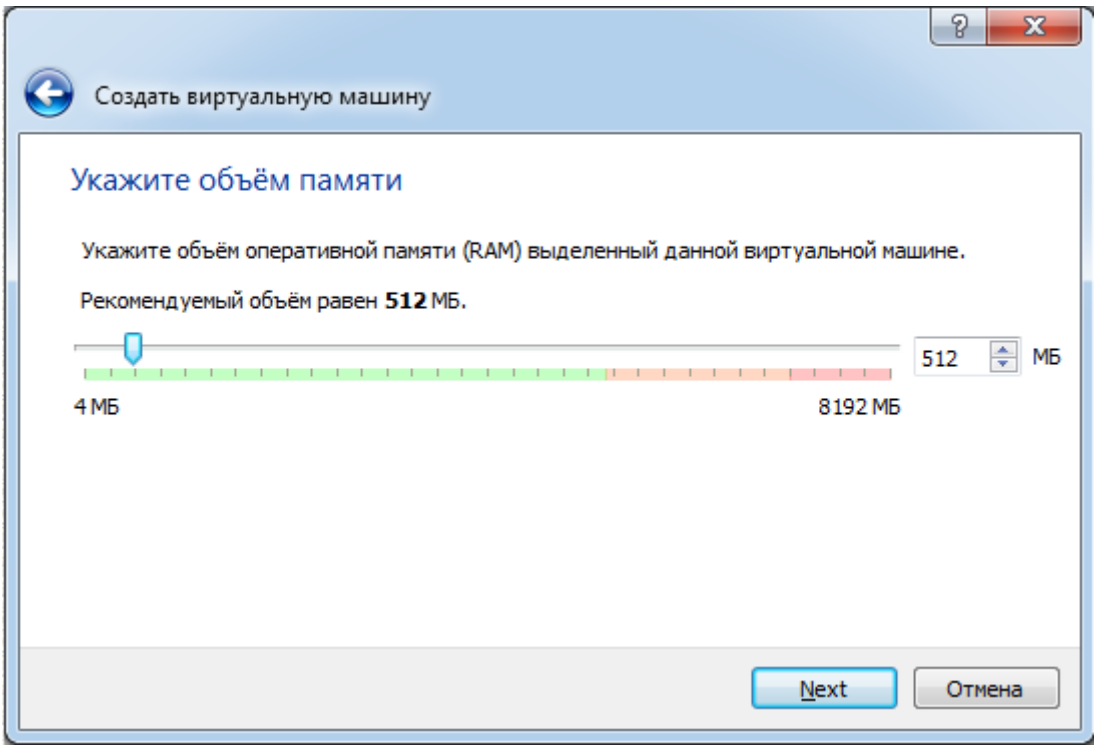


Рис. 3. Выделение объема оперативной памяти виртуальной машины

Следующим шагом формирования конфигурации аппаратной платформы виртуальной машины является создание жёсткого диска.

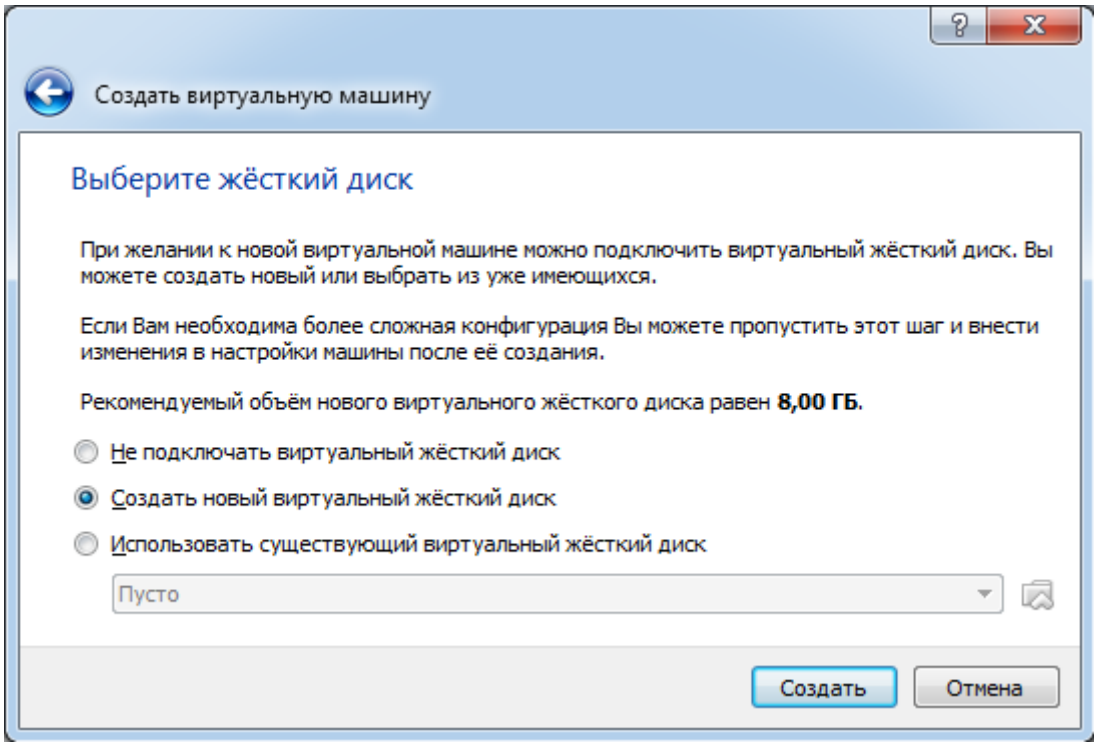


Рис. 4. Подключение жесткого диска виртуальной машины

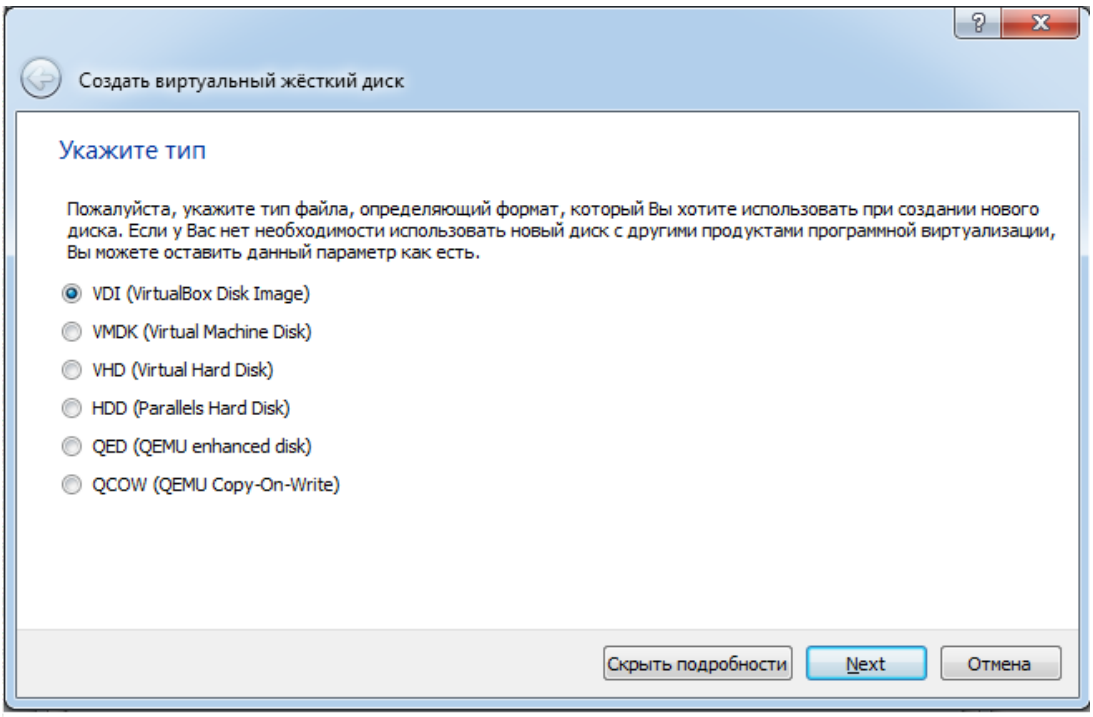


Рис. 5. Выбор типа жесткого диска для виртуальной машины

Далее мастер предложит выбрать способ управления объёмом жёсткого диска. Различают два способа: однократное создание жёсткого диска фиксированного (обычно максимально требуемого) объёма и динамическое расширение объёма диска по мере необходимости его заполнения. Выбор способа неоднозначен. При динамическом изменении объёма экономится место на жёстком диске хостовой машины, а при статическом (фиксированном) объёме – обеспечивается лучшая производительность операций ввода/вывода данных. На рисунке показан выбор динамического способа

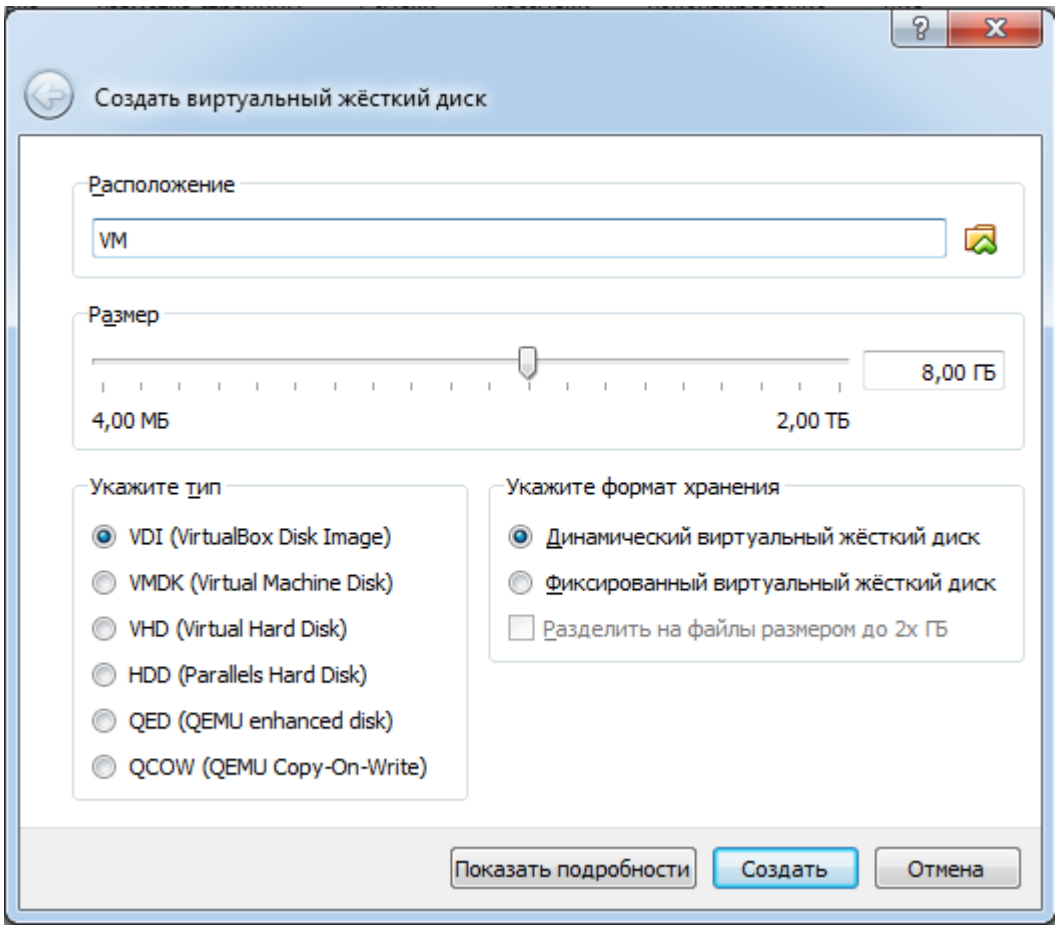


Рис. 6. Создание жесткого диска для виртуальной машины

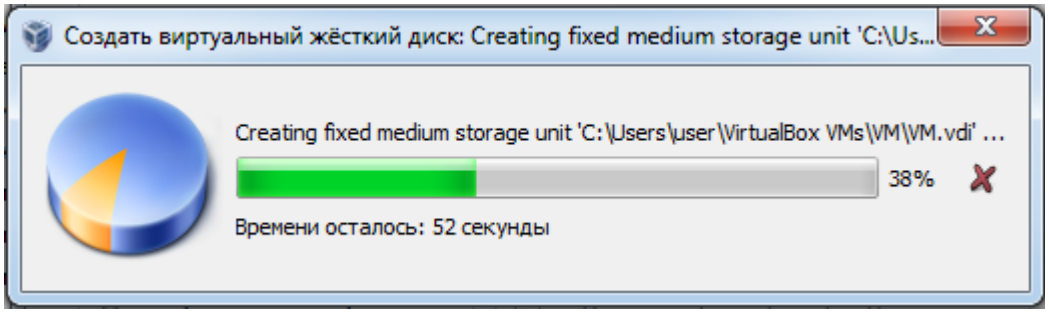


Рис. 7. Процесс создания жесткого диска

Таким образом, минимальная конфигурация гостевой виртуальной машины создана, и такая машина появится в списке машин, доступных менеджеру VirtualBox. Любую машину из указанного списка можно запустить на исполнение.

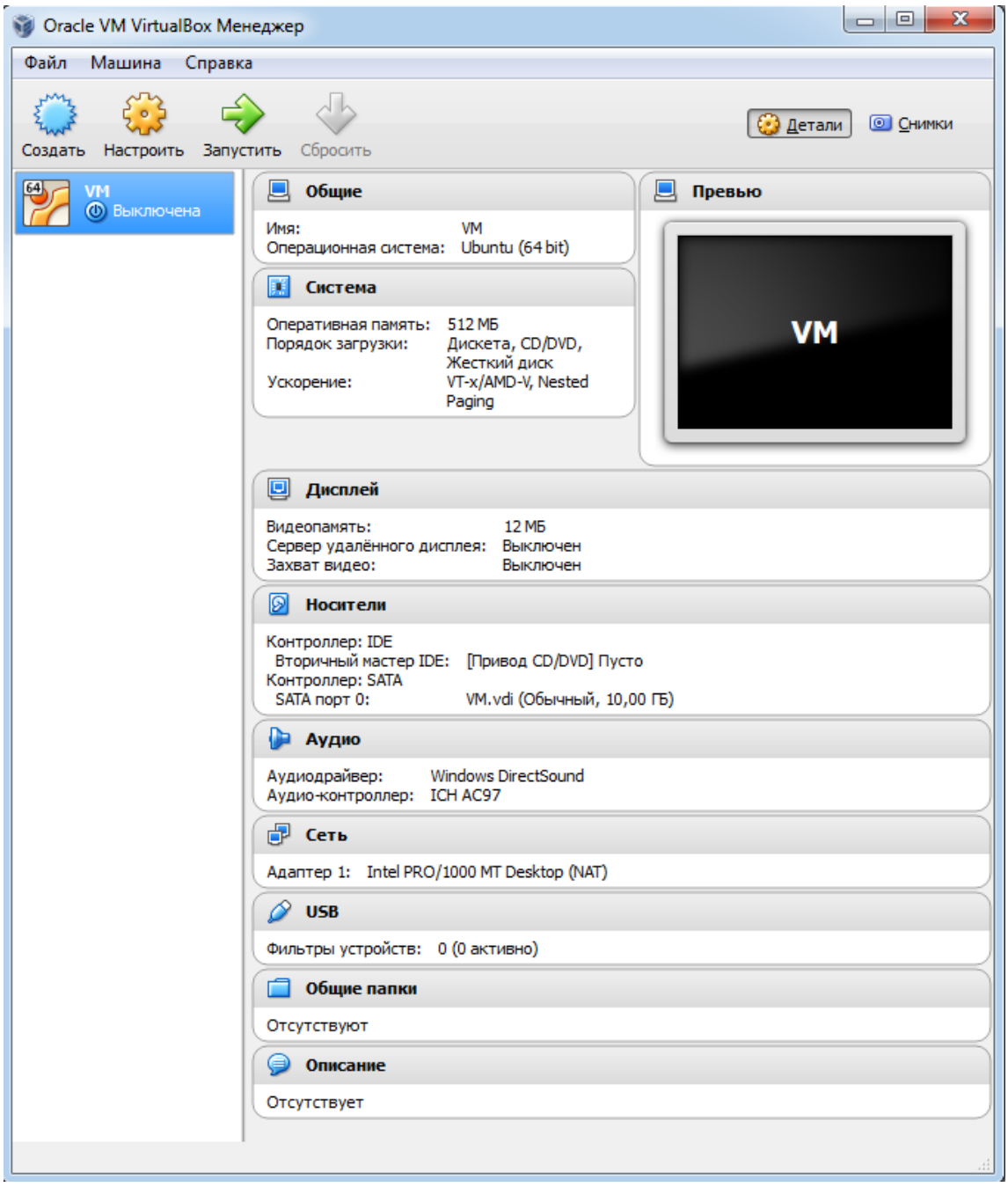


Рис. 8. Окно настроек виртуальной машины

В появившемся окне настроек следует выбрать группу настроек «Система», а внутри неё вкладку «Процессор».

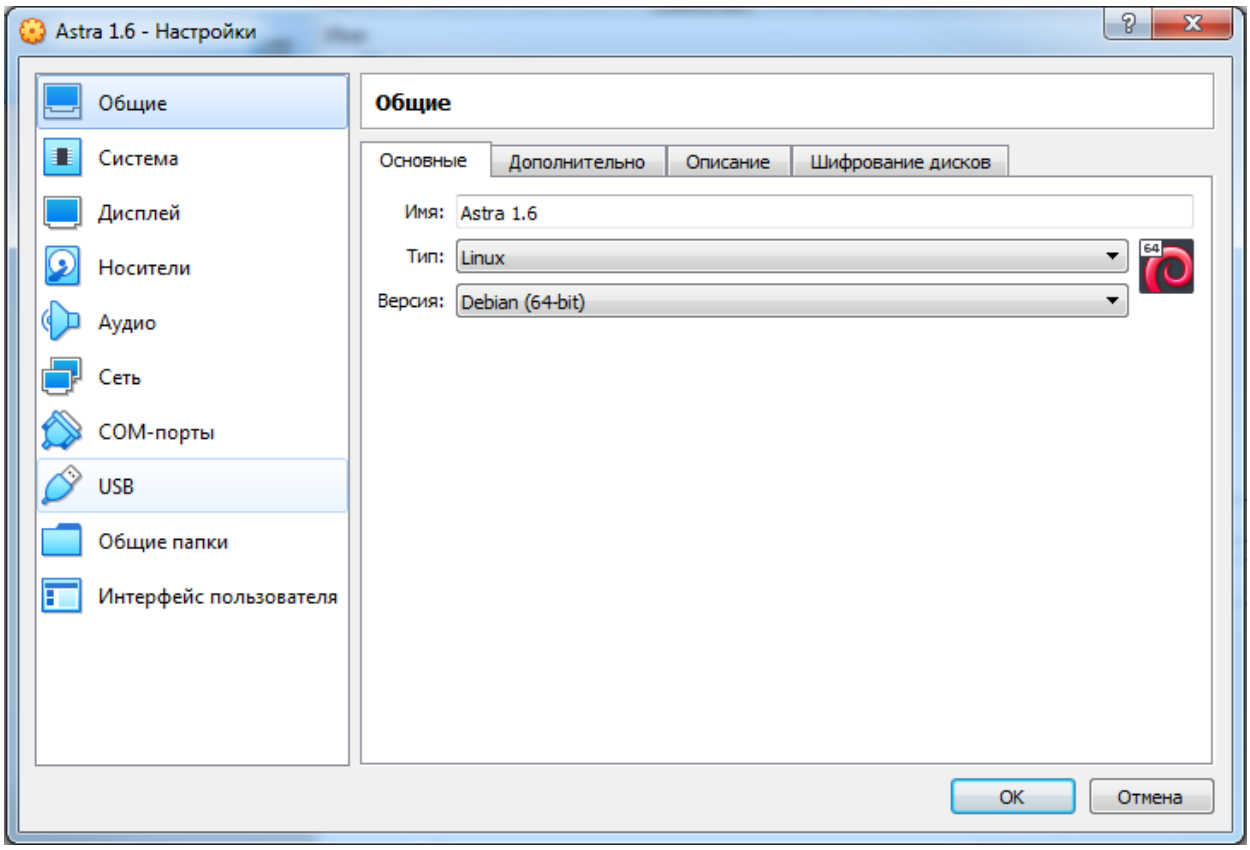


Рис. 9. Окно редактирования атрибутов виртуальной машины

Здесь можно выбрать, какое количество ядер процессора хостовой машины будет использовать процессор гостевой машины, а также включить PAE для поддержки больше 4 Гб RAM в 32 битных системах и режим эмуляции EFI. Затем следует перейдите на вкладку «Ускорение».

Ускорение будет обеспечено, если выбрать режим аппаратной виртуализации, а также включить такие дополнительные возможности, как AMD-V и Intel-VT. Это увеличит скорость работы и гостевой машины, и менеджера виртуальных машин.

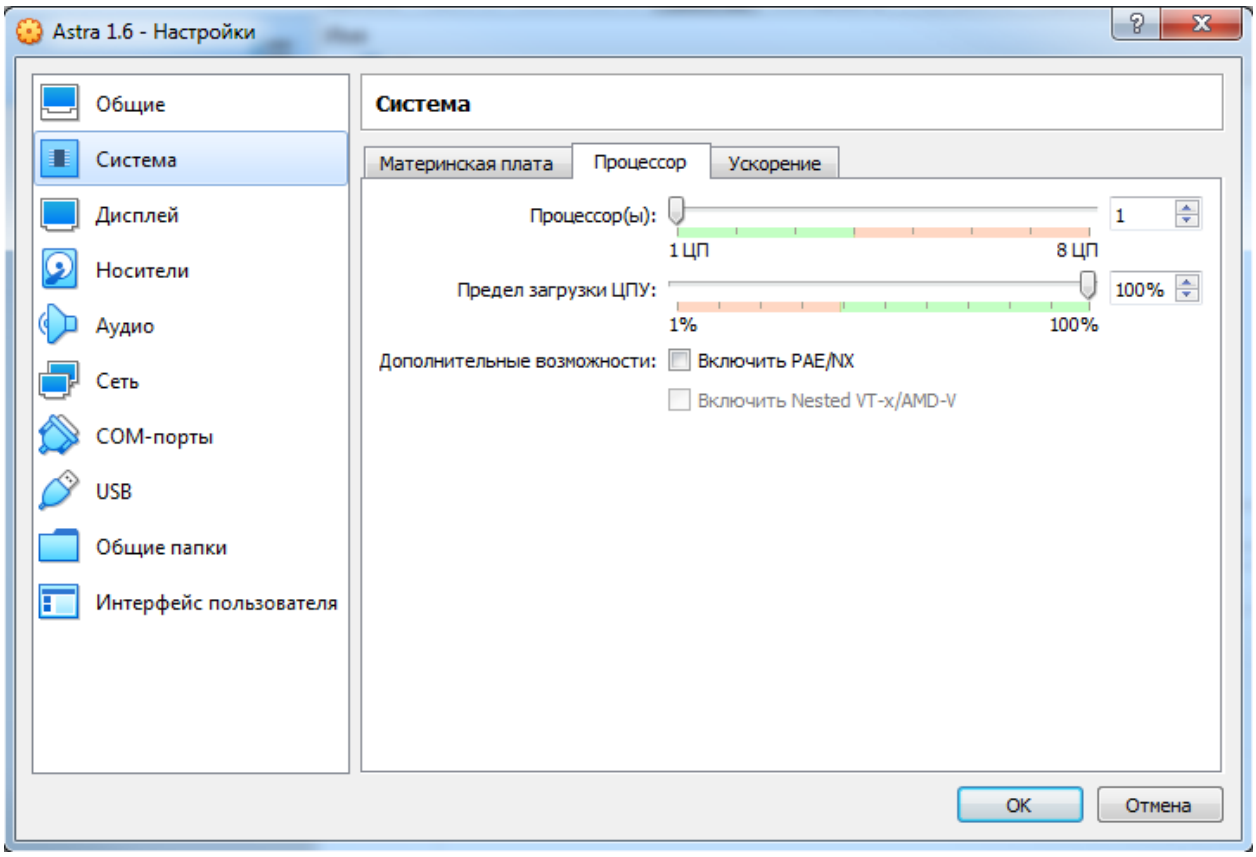


Рис. 10. Определение количества ядер процессора для виртуальной машины

По умолчанию для виртуальной машины доступно 18 Мб видеопамяти. Это очень мало для современных систем. Поэтому рекомендуется в настройках выбрать пункт меню «Дисплей» и, потянув ползунок «Видеопамять», увеличить объём видеопамяти гостевой машины до значения не менее 128 Мб. Также здесь можно настроить количество мониторов и другие параметры графики.

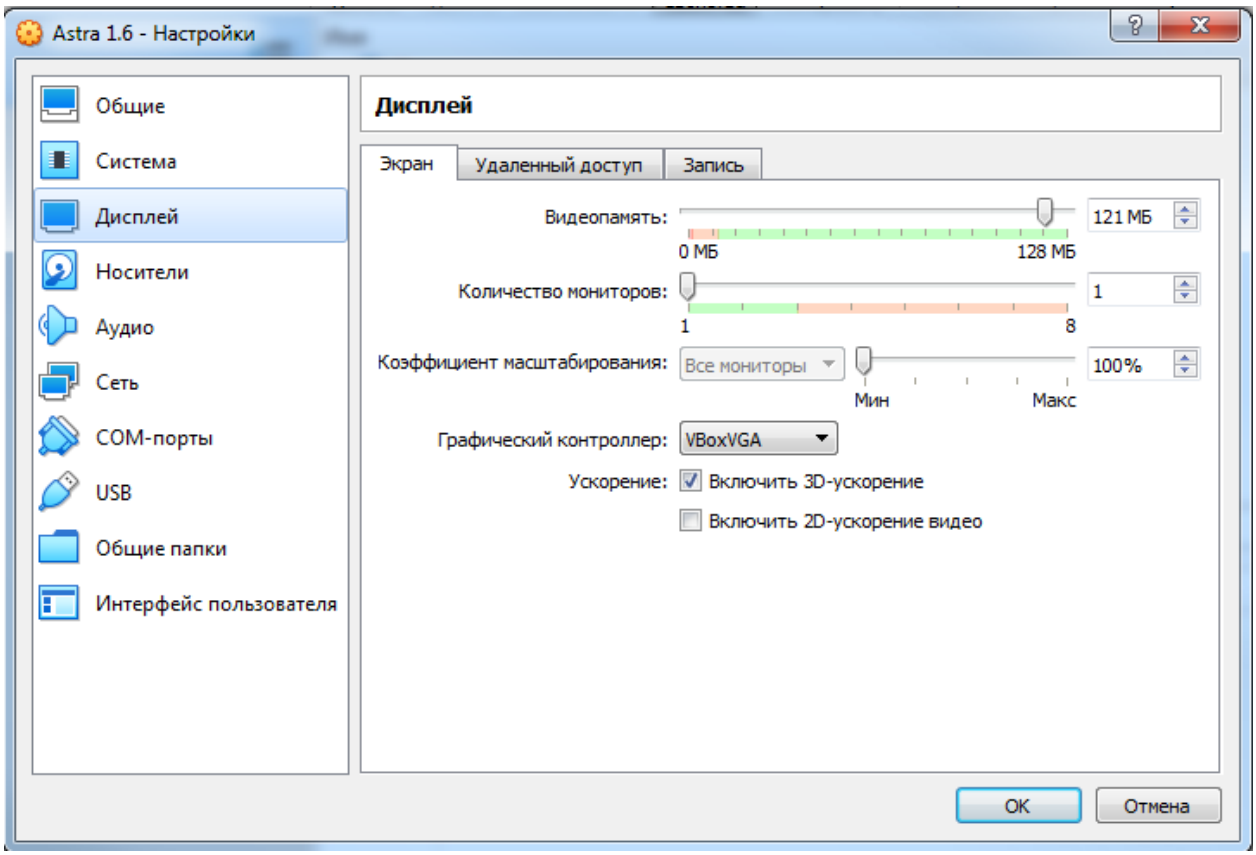


Рис. 11. Конфигурация видеопамяти для виртуальной машины

Одной из важных настроек для виртуальной машины является настройка сети. Для того, чтобы настроить сетевую конфигурацию необходимо перейти в раздел «Сеть» и в во вкладке «Тип подключения» выбрать «Сетевой мост», а также во вкладке «Имя» необходимо выбрать имя сетевого интерфейса (драйвера).

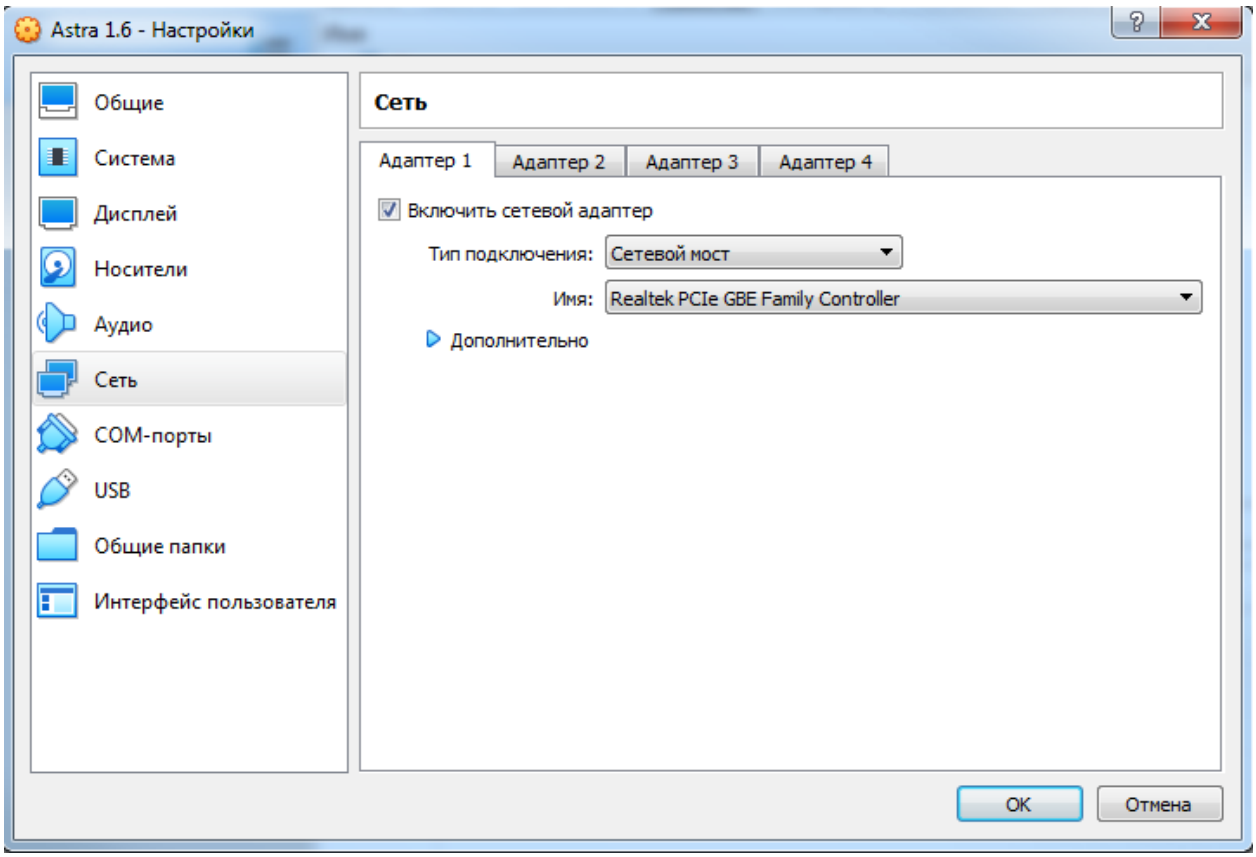


Рис. 12.Конфигурация сети для виртуальной машины

Перед запуском виртуальной машины следует добавить привод оптических дисков (iso-образ операционной системы)

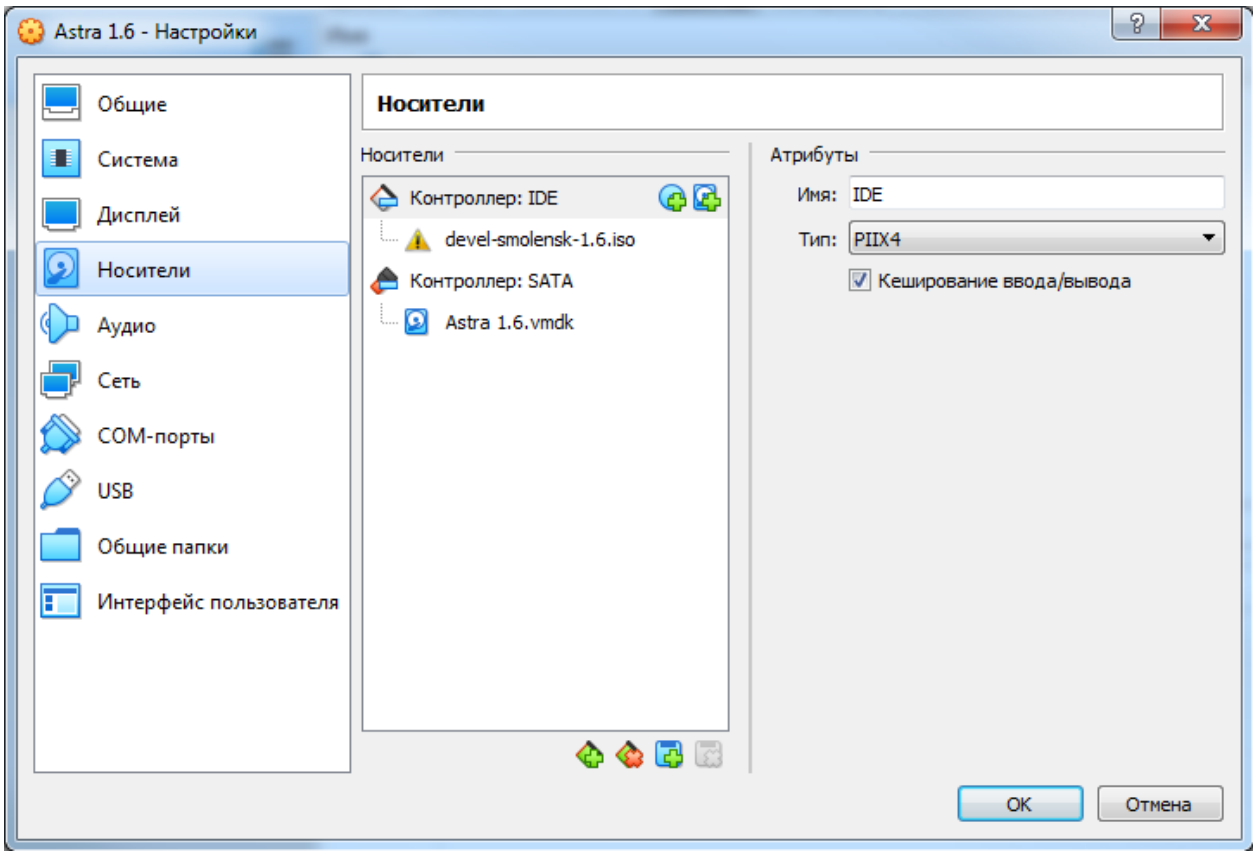


Рис. 13. Подключение оптических дисков к виртуальной машине

При первом запуске виртуальной машины (без установленной на неё операционной системы) необходимо выбрать носитель (если не был выполнен предыдущий шаг), содержащий дистрибутив с которого будет установлена новая система операционная система. Таким носителем может быть DVD-ROM или ISO образ системы. Также имеется возможность изменить этот образ из меню «Устройства» -> Оптические диски для уже запущенной гостевой машины

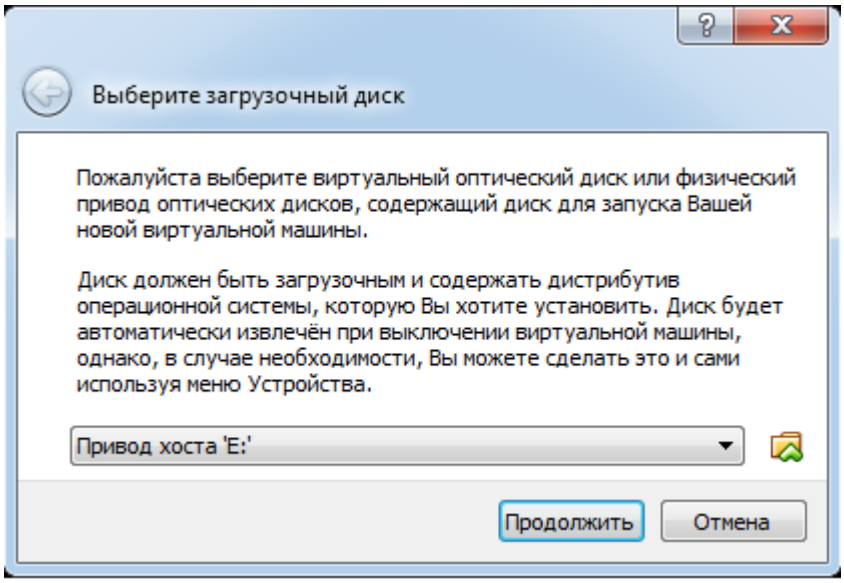
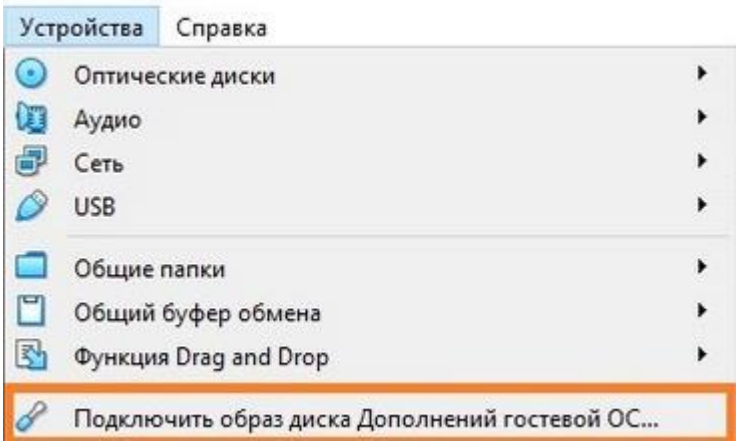


Рис. 14. Выбор загрузочного образа для установки операционной системы

Далее будет запущен процесс установки и следует выполнить шаги в зависимости от выбранного дистрибутива.

Раздел №4. Установка гостевых дополнений

Дополнения гостевой машины позволяют использовать такие возможности, как общий буфер обмена, общие папки, перетаскивание файлов, интеграция экрана, адаптация разрешения виртуальной машины и многое другое. Это необходимый набор программных библиотек для дальнейшей работы с виртуальными машинами. Дополнения устанавливаются в каждую гостевую систему и делают использование VirtualBox проще. Чтобы установить дополнения в меню «Устройства» следует выбрать «Подключить образ дополнений гостевой ОС».



Раздел №5. Заключение и выводы.

В данном разделе необходимо привести пронумерованный перечень выводов, которые были сделаны в результате установки средств виртуализации и виртуальной машины.

Установка и настройка систем управления проектами

Система управления проектами	
Назначение систем управления проектами	
Система управления проектами предназначена для структурирования и классификации задач, учета трудоемкости, временных затрат и наглядного представления состояния проектов в различных форматах. Доступ к системе базируется на ролевой модели, в соответствии с которой назначаются права пользования информационными ресурсами системы. Имеет гибкую базу расширений, за счет чего поддерживает интеграцию, в том числе, с системами оповещения и контроля версий.	
Ключевые аспекты	**Практический эффект**
• постановка и структурирование задач; • календарное и сетевое планирование; • формирование распоряжений и заявок (backlog проекта); • событийное оповещение о статусах задач; • определение зон ответственности каждого разработчика; • формирование досок задач; • версионирование документов; • формирование план-графиков; • взаимосвязь ревизий исходных кодов и задач; • многопользовательский ролевой доступ; • трекинг и типизация задач.	• контроль исполнения задач в рамках проектного подхода; • предиктивное планирование рабочего процесса; • контроль состояния проектов; • автоматизация расчетов фактической трудоемкости; • автоматизация формирования отчетов о проделанной работе; • контроль ревизий исходных кодов; • повышение степени формализованности и структурированности задач; • наглядный контроль исполнения технического задания; • контроль затрат чел./часов (по проектам и на рабочих местах).

Раздел №1. Введение.

Redmine — платформа с открытым кодом, предназначенная для управления проектами через веб-приложение. Платформа создана с использованием языка Ruby, обладает гибкими кросс-платформенными свойствами. Redmine является программным обеспечением с открытым исходным кодом, который распространяется на условиях лицензии GNU General Public License v2 (GPL).

Раздел №2. Установка и настройка системы управления проектами на базе программного продукта Redmine.

Установка Redmine предполагает пошаговую установку программных пакетов в среде GNU/Linux. В рамках данной лабораторной работы для роли «Руководитель проектов в области информационных технологий» достаточно выполнить команду:

sudo apt install redmine

Для роли «Специалист по информационным системам» в лабораторной работе установка Redmine представляет собой более углубленное погружение в данный процесс и предполагает реализацию следующих шагов.

1. Установить пакеты необходимые для работы redmine

*sudo apt install mc gcc zlib1g zlib1g-dev libssl-dev libyaml-dev *

*libcurl4-openssl-dev ruby libapache2-mod-passenger apache2 apache2-dev *

libapr1-dev libxslt1-dev libxml2-dev ruby-dev libmagickwand-dev imagemagick rails

2. Установить СУБД PostgreSQL 9.6

apt install postgresql

3. Установить пакет для разработчиков PostgreSQL 9.6

apt install postgresql-server-dev-9.6

4. Скопировать приложенную папку "redmine" в /home, чтобы получился путь /home/redmine

5. Запустить утилиту psql от имени пользователя postgres

sudo -u postgres psql postgres

6. Создать базу данных redmine, владельцем которой назначить postgres

//Блок актуален только в среде AstraLinux SE 1.6

//(иначе, необходимо назначить роль, у которой заданы мандатные атрибуты.
//Если владельцем базы данных будет назначена роль без заданных мандатных
//атрибутов, то при исполнении запросов к такой базе данных возникнет
//ошибка "не задан мандатный контекст")

CREATE DATABASE redmine WITH ENCODING='UTF8' OWNER=postgres;

Нажать CTRL + D для выхода из оболочки psql

7. Заменить файл /etc/postgresql/9.6/main/pg_hba.conf на /configs/pg_hba.conf находящийся в приложенной папке

 # В случае если файл /etc/postgresql/9.6/main/pg_hba.conf уже сконфигурирован для других нужд,

 # то добавить в него записи из приложенного файла /configs/pg_hba.conf:

local all postgres trust

local all all trust

host redmine postgres 127.0.0.1/32 trust

8. Скопировать файл /configs/database.yml из приложенной папки в директрию /home/redmine/redmine-4.0.5/config/

9. Перейти в директорию /home/redmine/redmine-4.0.5 и выполнить команды

bundle install --local

bundle exec rake generate_secret_token

RAILS_ENV=production bundle exec rake db:migrate

RAILS_ENV=production bundle exec rake redmine:load_default_data

10. Выдать права пользователю и группе www-data на соответствующие директории в каталоге /home/redmine

cd /home

sudo chown -R www-data:www-data /home/redmine

cd home/redmine/redmine-4.0.5

sudo chmod -R 755 files log tmp public/plugin_assets

sudo chown www-data:www-data Gemfile.lock

11. Открыть файл /etc/apache2/apache2.conf и отключить мандатный режим web-сервера Apache2, изменив строку "AstraMode on" на:

//Шаг актуален только в среде AstraLinux SE 1.6

Astra secure mode

AstraMode off

12. Файл /configs/master.conf в /etc/apache2/sites-available

Создать файл /etc/apache2/sites-available/master.conf со следующим содержимым:

*<VirtualHost *:80>*

ServerAdmin admin@redmine.ru # адрес электронной почты администратора

Servername proj.ru # желаемое доменное имя

 # Хост по-умолчанию для всех сайтов

 # Все остальные сайты поднимать через Symlinks внутри директории /var/www

DocumentRoot /var/www/

 # Настройки для запуска ruby, passenger для

 # системы управления проектами redmine в директории /var/www/redmine

 # Реальное расположение /home/redmine/redmine-4.0.5/public

<Directory /var/www/redmine>

AllowOverride All

Options FollowSymLinks

RailsEnv production

RackBaseURI /redmine

PassengerResolveSymlinksInDocumentRoot on

Options -MultiViews

</Directory>

 # Пример добавления другого сайта на данном хосте

 # Все директивы необходимо добавлять внутри раздела <Directory>!!

 # Для теста необходимо добавить раздел <Directory> для поддиректории /var/www/test,

 # сделать символическую ссылку test в /var/www/ на любую директорию

 # и перезапустить apache2 (systemctl reload apache2)

#<Directory /var/www/test>

 # AllowOverride All

 # Options FollowSymLinks

 # Options +Indexes

#</Directory>

 # По-умолчанию перенаправление на сайт redmine

RedirectMatch ^/$ /redmine/

 # Логгирование ошибок

ErrorLog ${APACHE_LOG_DIR}/error.log

CustomLog ${APACHE_LOG_DIR}/access.log combined

#</VirtualHost>

13. Добавить строку "PassengerUser www-data" (без кавычек) в файл /etc/apache2/mods-available/passenger.conf

14. Создать в директории /var/www/ символическую ссылку на папку /home/redmine/redmine-4.0.5/public

Вариант 1:

 # - зайти в MidnightCommander sudo mc

 # - перейти в директрорию /home/redmine/redmine-4.0.5 и установить указатель на папке public

 # - в другой панели MidnightCommander перейти в директорию /var/www/

 # - нажать клавишу F9 и выполнить Файл->Смиволическая ссылка

 # - в поле "Имя символической ссылки" ввести имя символической ссылки - /var/www/redmine

Вариант 2: выполнить команду

sudo ln -s /home/redmine/redmine-4.0.5/public/ /var/www/redmine

15. Отключить конфигурацию 000-default.conf, подключить конфигурацию master.conf и выполнить перезагрузку web-сервера Apache2

sudo a2dissite 000-default.conf

sudo a2ensite master.conf

sudo systemctl reload apache2

16. Протестировать работу: вбить ip-адрес (или доменное имя, если настроен доступ по DNS) в адресную строку браузера

Убедиться, что открылась стартовая страница Redmine. Авторизоваться под пользователем admin. Настроить для него новый пароль.

17. Резервирование Redmine (как БД, так и системы и пользовательских файлов)

Установить утилиту для копирования rsync

sudo apt-get install rsync

Запустить скрипт резервного копирования /home/redmine/backup_redmine.sh.

Скрипт будет делать резервные копии в директорию /home/redmine/backup.

При необходимости делать резервные копии на другую ЭВМ примонтировать к каталогу /home/redmine/backup директорию с другой машины. Для этого в файле /etc/fstab для пути /home/redmine/backup может быть примонтирован каталог из другой ЭВМ в локальной сети.

18.Установка темы Redmine

Скопировать содержимое архива в папке themes в директорию /home/redmine/"версия redmine"/public/themes.

Выбрать скопированную тему в настройках администратора в интерфейсе redmine.

Раздел №3. Функциональные возможности и применение системы управления проектами на базе программного продукта Redmine.

Функциональные возможности

Система управления проектами на базе Redmine должна быть доступна пользователям в рамках выделенного сегмента сети и предоставлять пользователям следующие возможности:

- ведение нескольких проектов;
- гибкая система доступа, основанная на ролях;
- система отслеживания ошибок;
- ведение новостей проекта, документов и управление файлами;
- оповещение об изменениях с помощью электронной почты;
- форумы для каждого проекта;
- учёт временных затрат;
- настраиваемые произвольные поля для инцидентов, временных затрат, проектов и пользователей;
- создание записей об ошибках на основе полученных писем;
- поддержка множественной аутентификации LDAP;
- возможность самостоятельной регистрации новых пользователей;
- многоязыковой интерфейс (в том числе русский);

Пользователи системы

Пользователи являются одним из центральных понятий предметной области. Модель пользователя является основой для идентификации и аутентификации работающего с системой.

Роли

Роли пользователей определяются гибкой моделью определения прав доступа пользователей. Роли включают в себя набор привилегий, позволяющих разграничивать доступ к различным функциям системы.

Пользователям назначается роль в каждом проекте, в котором он участвует, например «менеджер в проекте по разработке сайта А», «разработчик в проекте по поддержанию интранета компании» или «клиент в проекте по рефакторингу информационной системы компании Б». Пользователь может иметь несколько ролей. Назначение роли для отдельной задачи (issue) в данный момент невозможно.

Проекты

Проект является одним из основных понятий в предметной области систем управления проектами. Благодаря этой сущности возможно организовать совместную работу и планирование нескольких проектов одновременно с разграничением доступа различным пользователям (см. выше). Проекты допускают иерархическую вложенность.

Трекеры

Трекеры являются основной классификацией, по которой сортируются задачи в проекте. Само по себе понятие «трекер» восходит к системам учёта ошибок (англ. Bug tracking tool), представлявшим каждая в отдельности один проект. По сути, в системе трекеры представляют собой аналог подклассов класса «Задача» и являются основой для полиморфизма разного рода задач, позволяя определять для каждого их типа различные поля. Примерами трекеров являются «Улучшение», «Ошибка», «Документирование», «Поддержка».

Задачи

Задачи являются центральным понятием всей системы, описывающим некую задачу, которую необходимо выполнить. У каждой задачи в обязательном порядке есть описание и автор, в обязательном порядке задача привязана к трекеру.

Каждая задача имеет статус. Статусы представляют собой отдельную сущность с возможностью определения прав на назначение статуса для различных ролей (например, статус «отклонен» может присвоить только менеджер) или определение актуальности задачи (например, «открыт», «назначен» — актуальные, а «закрыт», «отклонен» — нет).

Для каждого проекта отдельно определяются набор этапов разработки и набор категорий задач. Среди других полей интересны также «оцененное время», служащее основой для построения управленческих диаграмм, а также поле выбора наблюдателей за задачей (см. «Получение уведомлений»). К задачам имеется возможность прикреплять файлы (имеется отдельная сущность «Приложение»). Значения других перечислимых свойств (например, приоритетность) хранятся в отдельной общей таблице.

Отслеживание изменения статуса задач

За отслеживание изменений параметров задач пользователями в системе отвечают две сущности: «Запись журнала изменений» и «Измененный параметр». Запись журнала отображает одно действие пользователя по редактированию параметров задачи и/или добавление комментария к ней. То есть служит одновременно инструментом ведения истории задачи и инструментом ведения диалога.

Сущность «Измененный параметр» привязана к отдельной записи журнала и предназначена для хранения старого и нового значения измененного пользователем параметра.

Связи между задачами

Задачи могут быть взаимосвязаны: например, одна задача является подзадачей для другой или предшествовать ей. Эта информация может быть полезна в ходе планирования разработки программы, за её хранение в Redmine отвечает отдельная сущность.

Учет затраченного на проект времени

Система поддерживает учет затраченного времени благодаря сущности «Затраченное время», связанной с пользователями и задачей. Сущность позволяет хранить затраченное время, вид деятельности пользователя (разработка, проектирование, поддержка) и краткий комментарий к работе. Эти данные могут быть использованы, например, для анализа вклада каждого участника в проект или для оценки фактической трудоемкости и стоимости разработки.

Привязка репозиториев

Redmine предоставляет возможность интеграции с различными системами контроля версий (репозиториями). Интеграция заключается в отслеживании изменений во внешнем репозитории, их фиксации в базе данных, анализе изменений с целью их привязки к определенным задачам. В инфологической структуре системы за интеграцию с внешними репозиториями отвечают три сущности: «Репозиторий», «Редакция» и «Изменение». «Репозиторий» представляет собой связанную с проектом сущность, хранящую тип подключенного репозитория, его местонахождение и идентификационные данные его пользователя.

«Редакция» является отображением редакции репозитория, и, кроме информационных полей, может быть привязана к конкретной задаче (для этого требуется указать в описании изменений «refs #NUM», где NUM – номер задачи), и к пользователю-автору редакции. Сущность «Изменение» предназначена для хранения списка измененных (добавленных, удаленных, перемещенных, модифицированных) файлов в каждой редакции.

Получение уведомлений

Уведомления пользователей об изменениях, происходящих на сайте, осуществляется с помощью сущности «Наблюдатели», связывающей пользователей с объектами различных классов (проекты, задачи, форумы и др.). В базе данных хранятся также ключи доступа к подписке RSS, позволяющие получать уведомления посредством этой технологии, также уведомления рассылаются с помощью электронной почты.

Администрирование системы управления проектами на базе Redmine

Задача администрирования системы управления проектами – это начало работы с системой. На данном шаге определяются ключевые характеристики модели работ по проекту (проектам), которые буду заведены в системе. Ниже приведены некоторые примеры атрибутов модели работ.

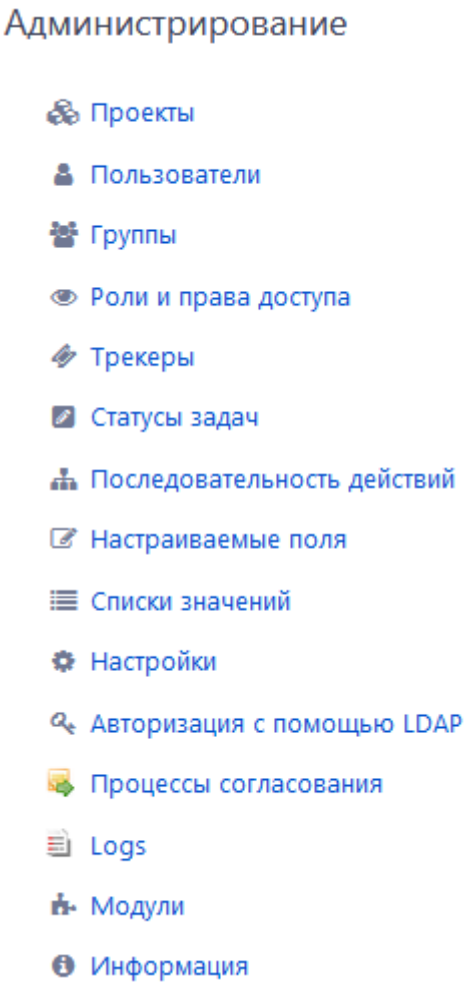


Рис. 15. Вкладка «Администрирование» системы Redmine

Роль определяет права доступа и набор возможных действий конкретного пользователя. Каждому пользователю может быть назначено несколько ролей. Например.

1. Руководитель проекта. Имеет полный доступ и перечень полномочий в проекте. Основная задача руководителя состоит формулировании задачи проекта, контроль за выполнением работ ключевых (Гл. и Вед. инженеров проекта) сотрудниками.
2. Главный инженер проекта. Имеет урезанный доступ и перечень полномочий в проекте. Основная задача главного инженера состоит в

контроле выполнения задачи проекта, контроль за выполнением работ ключевых сотрудниками.

Трекеры:

- Разработка.
- Документирование.
- Определение.
- Согласование.
- Утверждение.
- Уточнение.
- Проектирование.
- Алгоритмизация.
- Моделирование.
- Испытания.
- Конфигурирование.
- Исследование.
- Интеграция.
- Отладка.
- Тестирование.
- Сопровождение.
- Корректировка.
- Модернизация.
- Адаптация.

Статусы и последовательности действий определяют жизненный цикл задачи.

- Запланирована.
- Назначена.
- В работе.
- Выполнена.
- Отменена.
- На контроле.
- Возвращена.
- Проверена.
- Закрыта.

Последовательность действий » Краткое описание

	Административный руководитель
Проект	164
Разработка КД	164
Разработка ОРД	164
Разработка ЭД	164
Разработка ПО	164
Закупка ПКИ	164
Закупка ПО	164
Согласование с ВП	164
Согласование с заказчиком	164
Согласование с ОТК	164
Наладка	164
Изготовление	164
Сборка	164
Предъявление ВП	164
Предъявление ОТК	164
Предварительные испытания	164
Предъявительские испытания	164
Приемосдаточные испытания	164
Упаковка	164
Отгрузка	164

Рис. 16. Окно редактирования последовательности действий Redmine

Создание проекта в Redmine

При создании проекта заполняются поля:

- имя проекта (необходимо указать имя соответствующее выполняемому варианту);
- описание проекта (должно быть приведено описание раскрывающее назначение будущей разработки);

– уникальный идентификатор;
– стартовая страница;
– «общедоступный» обозначает видимость только для включенных участников;

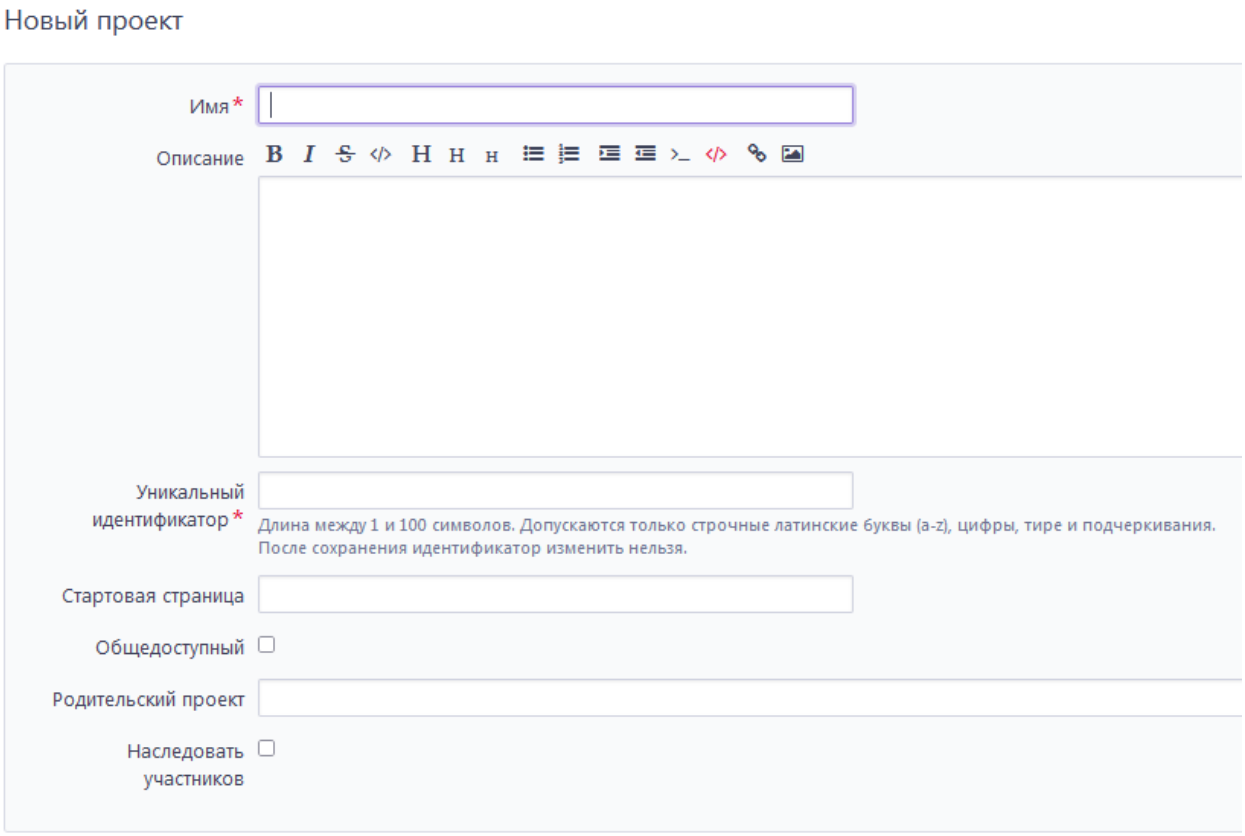


Рис. 17. Создание проекта в Redmine

Создание задач в Redmine

При создании задачи заполняются поля:

– трекер (ошибка, улучшение, поддержка);
– частная, заполняется в том случае, если нужно чтобы задачу видел только автор задачи и участник проекта, кому она назначена;
– тема (название задачи);
– описание (пояснение к задаче, если в теме недостаточно раскрыто что нужно сделать, каким-образом нужно решить задачу, комментарии к задаче);
– при создании задачи статус остается на значении Новая;
– приоритет, указывается приоритет, в котором задача должна быть решена;

– назначена – участник проекта, который будет заниматься решением задачи;
– версия – выбор, какому разделу в рамках проекта относится задача;
– файлы (можно добавить файлы, относящиеся к задаче, способу ее решения);
– родительская задача, если создаваемая задача является подзадачей, то в поле «родительская задача» указывается номер главной задачи;
– начата, указывается дата начала выполнения задачи;
– дата выполнения, указывается дата завершения выполнения задачи;
– оценка времени (можно не заполнять), указывается количество часов, которое планируется на выполнение задачи;
– готовность, указывается процент, на который задача уже выполнена (например, задача в которой нужно что-то доработать, можно указать процент, на который доработка уже сделана);
– наблюдатели, участники проекта, которые будут отслеживать выполнение задачи. Даже флажок установлен на Частная, наблюдатели будут видеть ход решения задачи, им будет приходить обновления по задаче.

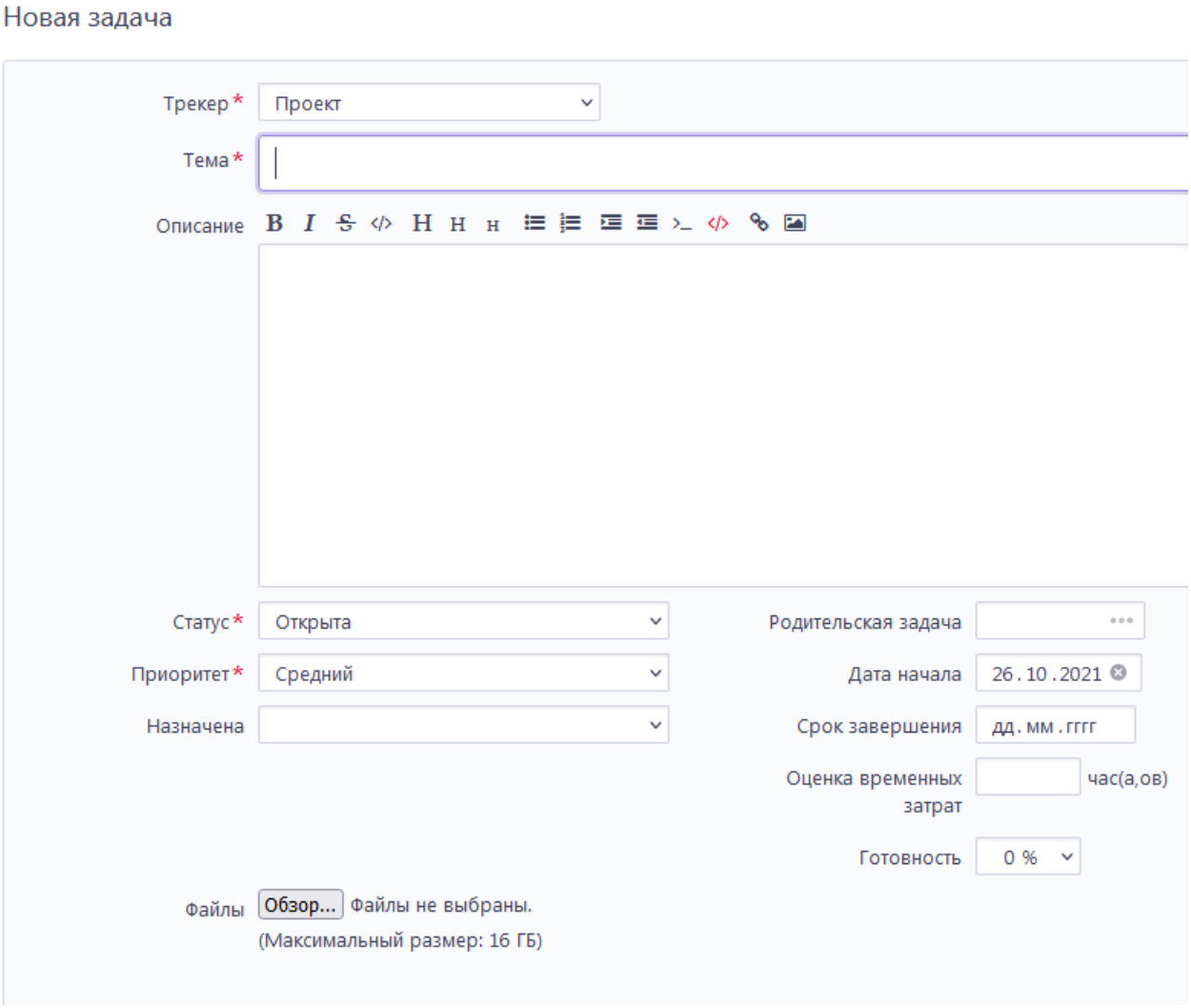


Рис. 18. Создание задачи в Redmine

Диаграмма Ганта

Диаграмма Ганта позволяет отслеживать сроки выполнения задачи. Слева находится перечисление задач, справа сроки. На самой диаграмме также находится горизонтальная красная пунктирная линия, которая обозначает текущий день. У задач находящихся за этой чертой, прошел срок выполнения.

На Диаграмме Ганта также присутствуют фильтры аналогичные в Задачах.

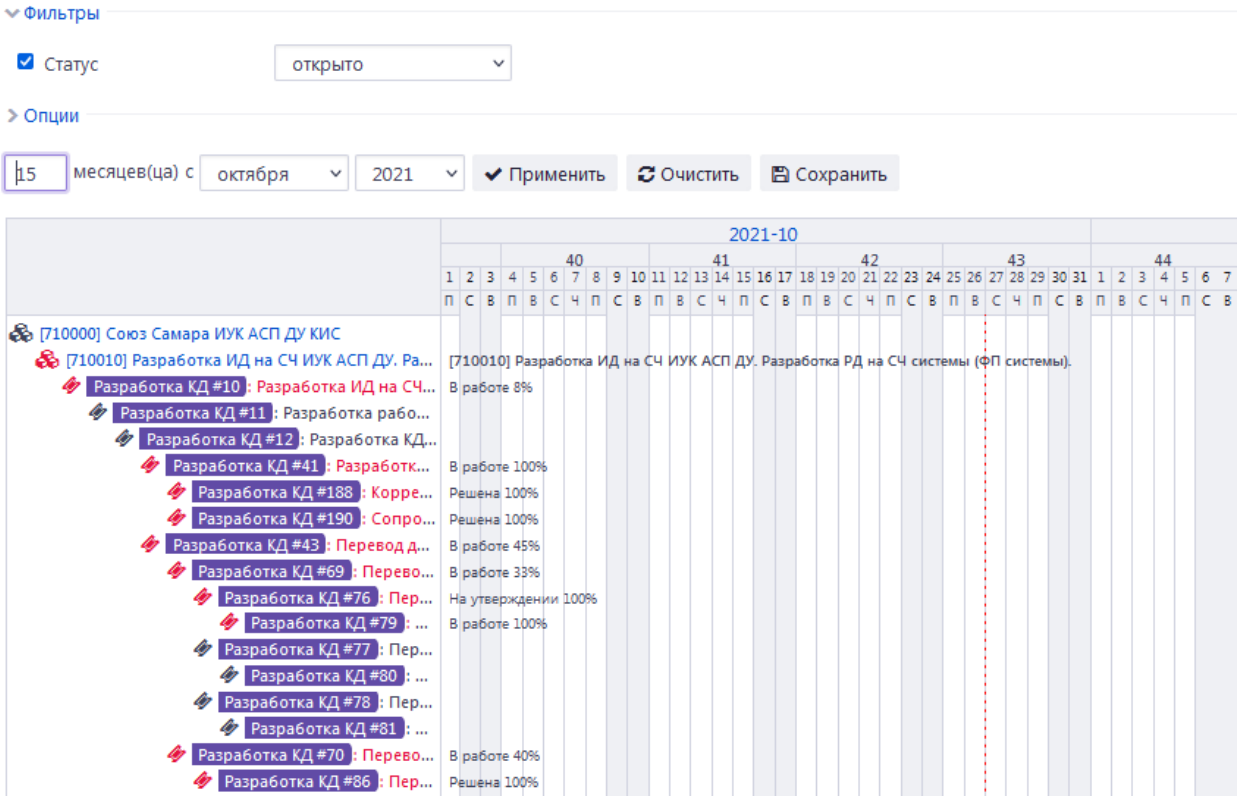


Рис. 19. Диаграмма Ганта в Redmine

Раздел №4. Заключение и выводы.

В данном разделе необходимо привести пронумерованный перечень выводов, которые были сделаны в результате установки системы управления проектами, формирования модели работ, распределения ролей и создания дерева целей и задач.

Установка и настройка систем контроля версий

Система контроля версий исходных кодов и хранения версий ПО	
Назначение систем контроля версий	
Система контроля версий исходных кодов и хранения версий ПО предназначена для организации централизованного хранилища исходных кодов и ПО, позволяющего отслеживать всю хронологию разработки ПО на всех этапах его жизненного цикла. Доступ к такому хранилищу может осуществляться на базе единых авторизационных механизмов, благодаря которым осуществляется разграничение доступа в соответствии с ролями и полномочиями. Имеется поддержка интеграции с системами развертывания, конфигурирования и системами управления проектами.	
Ключевые аспекты	**Практический эффект**
• централизованное хранение и управление исходными кодами; • веб-интерфейс управления репозиториями; • организация ветвления и управление коллективной разработкой; • создание «отсечек» и релизных версий; • хранение и версионирование релизов; • контроль «коммитов» и применение «Hooks»; • централизованное создание релизов.	• повышение надежности управления и контроля исходных кодов; • ретроспектива релизных и промежуточных версий ПО; • повышение качества исходных кодов; • облегчение процедур повторного использования кода.

Раздел №1. Введение.

В настоящее время сложность промышленных автоматизированных/информационных систем и приложений достигла такого уровня, при котором организация работ по разработке таких систем приобретает высокую сложность, что требует вовлечения в процесс целые коллективы разработчиков. При организации таких коллективов основные задачи заключаются в распределении обязанностей в рамках сформированных групп и обеспечении контроля выполнения задач в процессе всего цикла разработки.

На сегодняшний день существуют инструменты для ведения истории изменений и коллективной разработки программного кода, именуемые системами контроля версий.

Системы контроля версий подразделяются на локальные, централизованные и распределенные.

Раздел №2. Локальные, централизованные и распределенные системы контроля версий.

Локальный вид систем применяется для частного применения и только в рамках локального компьютера. Одной из популярных локальных систем контроля версий контроля была система контроля редакций (Revision Control System, RCS), которая до сих пор работает на многих компьютерах. Даже в популярной операционной системе Mac OS X разработчикам доступна команда rcs. RCS сохраняет на диске в специальном формате набор всех внесенных изменений (то есть разницу между файлами). В будущем это позволяет воссоздать любой файл в любой момент времени, добавив к нему все изменения.

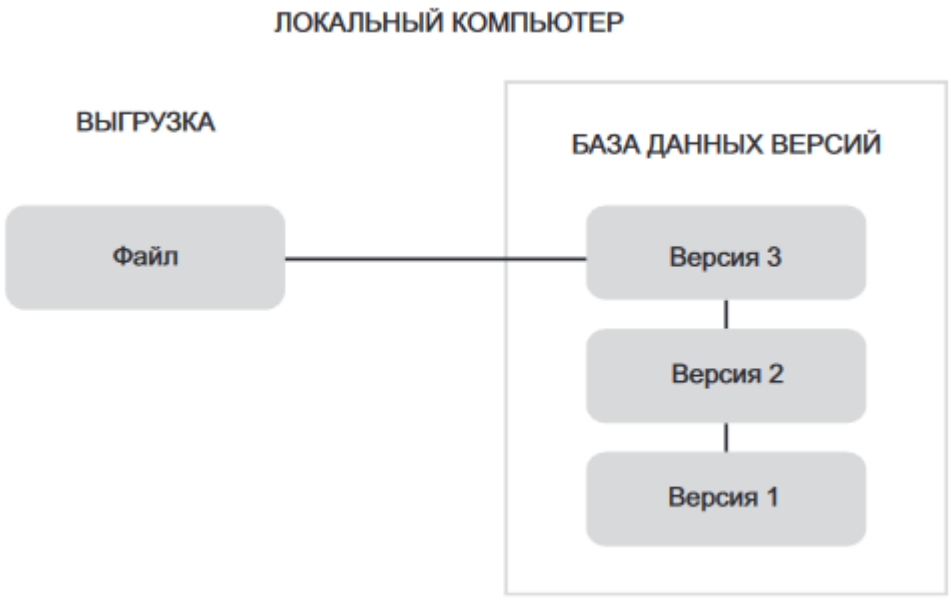


Рис. 20. Схема работы систем контроля версий локального вида

Централизованные системы контроля версий применяются для взаимодействия между многими разработчиками

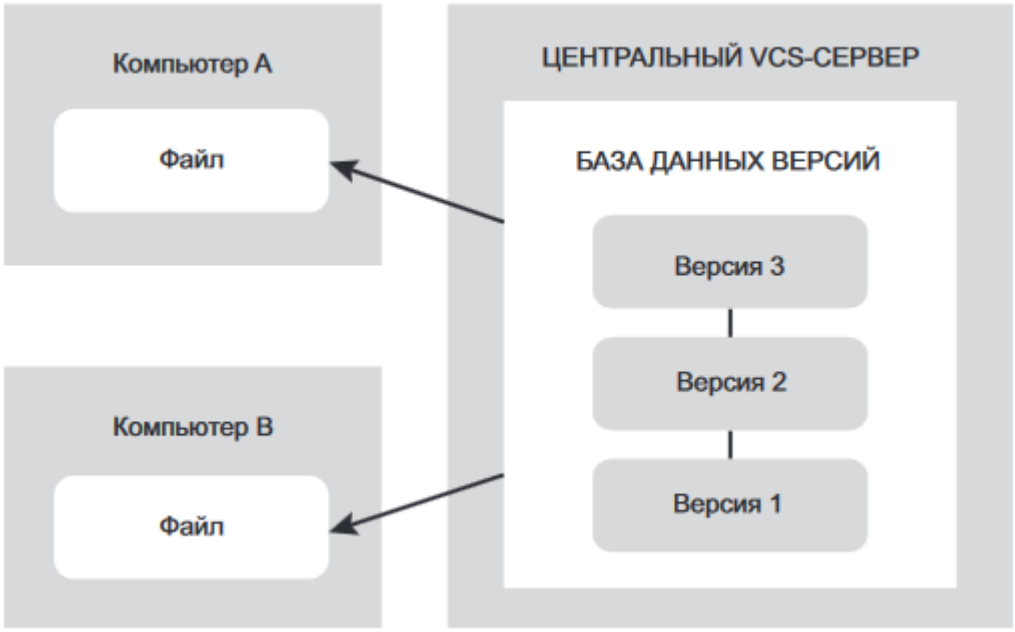


Рис. 21. Схема работы централизованных систем контроля версий

Такая схема имеет много преимуществ, особенно перед локальными системами контроля версий. Например, каждый человек, работающий над проектом, до определенной степени знает, чем занимаются его коллеги. Администраторы могут детально контролировать права допуска прочих сотрудников; администрировать CVCS намного проще, чем управлять локальными базами данных на каждом клиенте.

Однако есть у этой схемы и серьезные недостатки. Самым очевидным является единая точка отказа, представленная центральным сервером. Отключение этого сервера на час означает, что в течение часа любые взаимодействия невозможны. То есть вы не сможете сохранять вносимые изменения. При повреждении жесткого диска центральной базы данных и отсутствии нужных резервных копий теряется вся информация — вся история разработки проекта за исключением единичных снимков состояния, которые могут остаться на локальных компьютерах пользователей. Впрочем, та же самая проблема характерна и для локальных систем контроля версий — храня историю разработки проекта в одном месте, вы рискуете потерять все.

Именно здесь на первый план выходят ***распределенные*** системы контроля версий (Distributed Version Control System, DVCS). В DVCS (к примеру, Git, Mercurial, Bazaar или Darcs) клиенты не просто выгружают последние снимки файлов, они создают полную зеркальную копию репозитория. Соответственно в случае выхода из строя одного из серверов его работоспособность можно восстановить, скопировав один из клиентских репозиториев. Каждая такая выгрузка сопровождается полным резервным копированием всех данных.

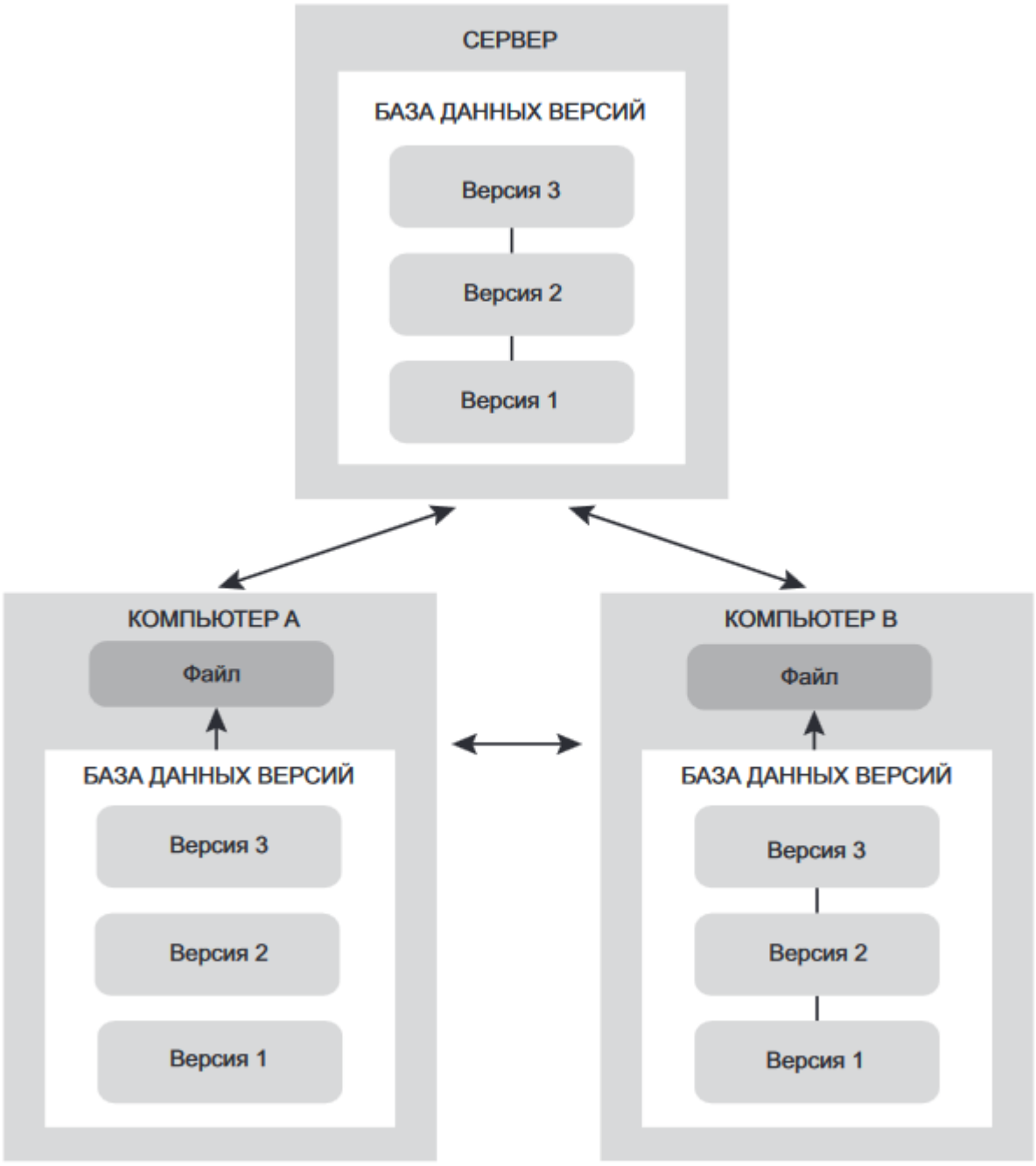


Рис. 22. Схема работы распределенных систем контроля версий

Более того, многие из этих систем обладают несколькими удаленными репозиториями, что позволяет разным рабочим группам сотрудничать в рамках одного проекта. Допустима настройка разных типов рабочих процессов, невозможная в централизованных системах, например в иерархических моделях.

Раздел №3. Основы работы с системой контроля версий Git. Установка и настройка Git.

Целеполагание и характеристики системы Git:

- быстродействие;
- простое проектное решение;
- мощная поддержка нелинейной разработки (тысячи параллельных ветвей);
- полностью распределенная система;
- возможность эффективной (в плане быстродействия и объема данных) работы с большими проектами, например, проект GNU/Linux.

Снимки против изменений

Главным отличием Git от любой другой системы контроля версий (в том числе Subversion и ей подобных) является восприятие данных. Большинство систем хранит информацию в виде списка изменений, связанных с файлами. Эти системы (CVS, Subversion, Perforce, Bazaar и т. п.) рассматривают хранимые данные как набор файлов и изменений, которые вносились в эти файлы в течение их жизни.

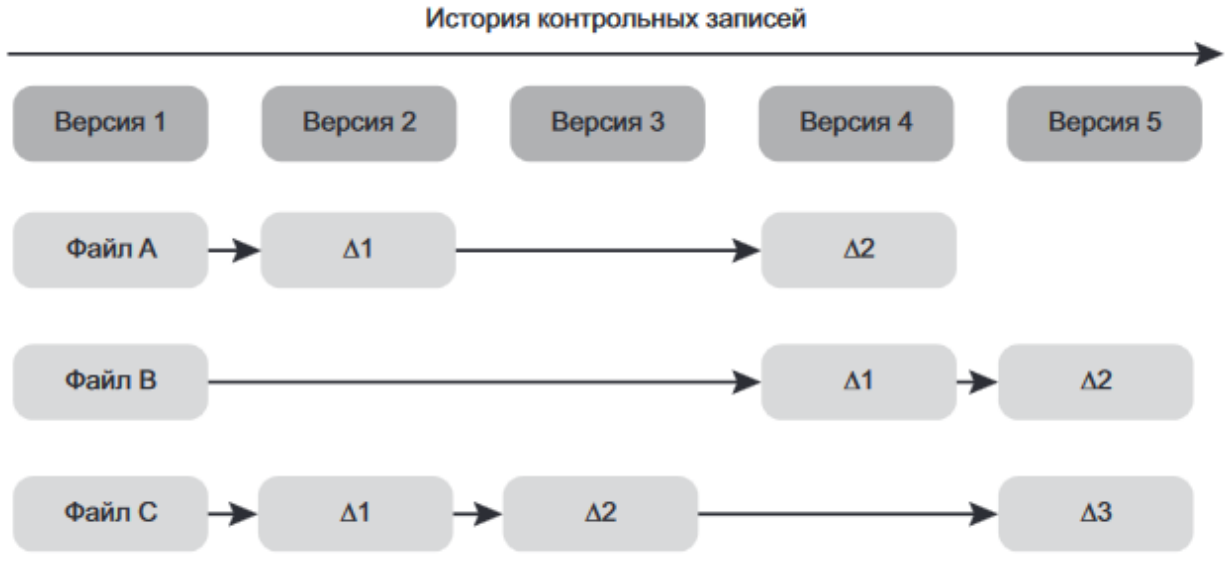


Рис. 23. История контрольных записей (1)

Система Git не воспринимает и не хранит файлы подобным образом. В ее восприятии данные представляют собой набор снимков состояния миниатюрной файловой системы. Каждый раз, когда вы создаете новую

версию или сохраняете состояние проекта в Git, по сути, делается снимок всех файлов в конкретный момент времени и сохраняется ссылка на этот снимок. Для повышения продуктивности вместо файлов, которые не претерпели изменений, сохраняется всего лишь ссылка на их ранее сохраненные версии. Git воспринимает данные скорее как поток снимков состояния (stream of snapshots).

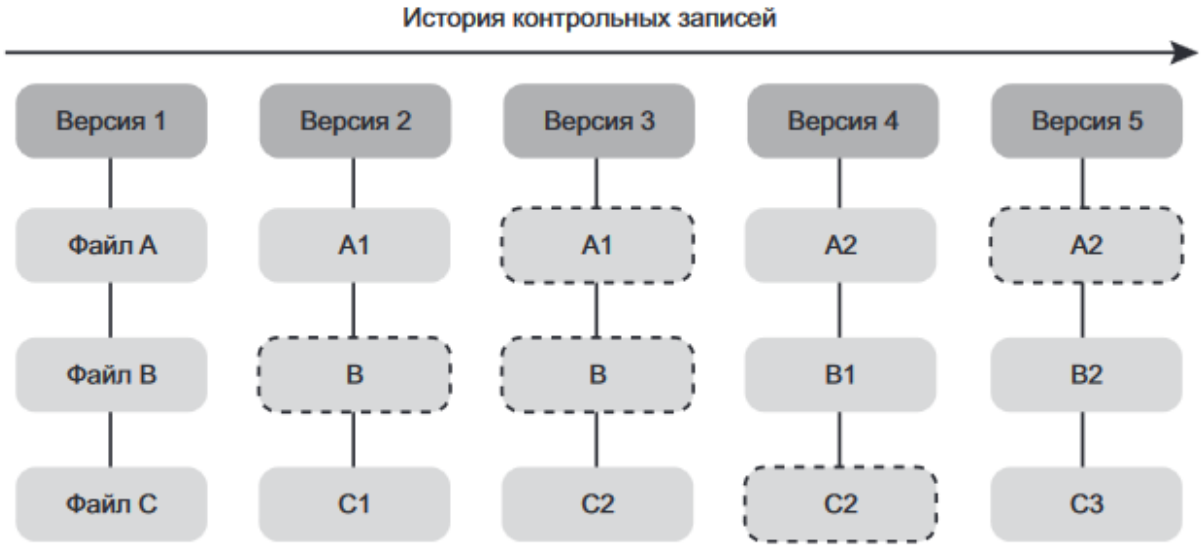


Рис. 24. История контрольных записей (2)

Это важное отличие Git почти от всех остальных VCS. И именно из-за него в Git приходится пересматривать практически все аспекты управления версиями, которые остальные системы скопировали у своих предшественниц. В результате Git больше напоминает не простую систему контроля версий, а миниатюрную файловую систему с удивительно мощным инструментарием.

Целостность Git

В системе Git для всех данных перед сохранением вычисляется контрольная сумма, по которой они впоследствии ищутся. То есть сохранить содержимое файла или папки таким образом, чтобы система Git об этом не узнала, невозможно. Эта функциональность встроена в Git на самом низком уровне и является неотъемлемым принципом ее работы. Невозможно потерять информацию или повредить файл скрытно от Git.

Механизм, которым пользуется Git для вычисления контрольных сумм, называется хешем SHA-1. Это строка из 40 символов, включающая в себя числа в шестнадцатеричной системе (0-9 и a-f) и вычисляемая на основе содержимого файла или структуры папки в Git. Хеш SHA-1 выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Вы будете постоянно наталкиваться на эти хеш-значения, так как Git использует их повсеместно. По сути, Git сохраняет данные в базе не по именам файлов, а по хешу их содержимого.

Надежность Git

Практически все операции в Git приводят к добавлению данных в базу. Систему сложно заставить выполнить неотменяемое действие или каким-то образом стереть данные. Как и при работе с любой VCS, незафиксированные данные можно потерять или повредить; но как только снимок состояния зафиксирован, потерять его становится крайне сложно, особенно если вы регулярно копируете базу в другой репозиторий.

Это делает работу с Git крайне комфортной, так как можно экспериментировать, не опасаясь серьезно навредить проекту.

Три состояния

Файлы в Git могут находиться в трех основных состояниях: зафиксированном, модифицированном и индексированном. Зафиксированное (committed) состояние означает, что данные надежно сохранены в локальной базе. Модифицированное (modified) состояние означает, что изменения уже внесены в файл, но пока не зафиксированы в базе данных. Индексированное (staged) состояние означает, что вы пометили текущую версию модифицированного файла как предназначенную для следующей фиксации.

Рис. 24. Три состояния файлов в Git

Папка Git – это место, где Git хранит метаданные и объектную базу данных проекта. Это наиболее важная часть Git, которая копируется при дублировании репозитория (хранилища) с другого компьютера.

Рабочая папка – это место, куда выполняется выгрузка одной из версий проекта. Эти файлы извлекаются из сжатой базы данных в папке Git и помещаются на жесткий диск вашего компьютера, готовые к использованию или редактированию.

Область индексирования – это файл, обычно находящийся в папке Git и хранящий информацию о том, что именно войдет в следующую операцию фиксации. Иногда ее еще называют промежуточной областью.

Базовый рабочий процесс в Git выглядит так:

1. Вы редактируете файлы в рабочей папке.
2. Вы индексируете файлы, добавляя их снимки в область индексирования.
3. Вы выполняете фиксацию, беря файлы из области индексирования и сохраняя снимки в папке Git.

Установка системы Git

Если вы хотите установить Git в операционной системе Linux при помощи установщика бинарных пакетов, в общем случае можно воспользоваться инструментом управления пакетами, входящим в имеющийся у вас дистрибутив. Например, в дистрибутиве Fedora (CentOS, RadHat) применяется утилита yum:

```
$ yum install git
```

Если вы пользуетесь дистрибутивом семейства Debian, например Ubuntu, попробуйте утилиту apt-get:

```
$ apt-get install git
```

Первичная настройка системы Git

Теперь, когда на вашей машине установлена система Git, нужно настроить ее окружение. На каждом компьютере эта процедура проводится только один раз; при обновлениях все настройки сохраняются. Впрочем, вы можете изменить их в любой момент, снова воспользовавшись указанными командами. Система Git поставляется с инструментом git config, позволяющим получать и устанавливать переменные конфигурации, которые задают все аспекты внешнего вида и работы Git. Эти переменные хранятся в разных местах.

1. Файл /etc/gitconfig содержит значения, действующие для всех пользователей системы и всех их репозиториев. Указав параметр --system при запуске git config, вы добьетесь чтения и записи для этого конкретного файла.
2. Файл ~/.gitconfig или ~/.config/git/config связан с конкретным пользователем. Чтение и запись для этого файла инициируются передачей параметра --global.
3. Параметры конфигурационного файла в папке Git (то есть .git/config) репозитория, с которым вы работаете в данный момент, действуют только на конкретный репозиторий.

При установке Git первым делом следует указать имя пользователя и адрес электронной почты. Это важно, так как данную информацию Git будет включать в каждую фиксируемую вами версию, и она обязательно включается во все создаваемые вами коммиты (зафиксированные данные):

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

Передача параметра --global позволяет сделать эти настройки всего один раз, так как в этом случае Git будет использовать данную информацию для всех ваших действий в системе. Если для конкретного проекта требуется указать другое имя или адрес электронной почты, войдите в папку с проектом и выполните эту команду без параметра --global.

Многие GUI-инструменты помогают выполнять эти действия при своем первом запуске.

Создание репозитория в существующем каталоге

Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в каталог проекта и в командной строке ввести

git init

Запоминать эти команды не нужно – они показываются в GIT-е при создании нового репозитория.

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория – основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем.

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

git add .

git commit -m "initial project version"

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

Клонирование существующего репозитория

Если вы хотите получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда `git clone`.

Клонирование репозитория осуществляется командой git clone [url]. Для примера склонируем репозиторий с этими лекциями:

git clone https://github.com/kolei/oap

Эта команда создаёт каталог с именем *OAP*, инициализирует в нём каталог .git, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новый каталог *OAP*, вы увидите в нём файлы, пригодные для работы и использования.

Если вы хотите клонировать репозиторий в каталог, отличный от *OAP*, то можно это указать в следующем параметре командной строки:

git clone https://github.com/kolei/oap mydir

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван `mydir`.

Запись изменений в репозиторий

Итак, у вас имеется Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать “снимки” состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить (обычно рекомендуют фиксировать каждое атомарное изменение, т.е. функцию, класс или законченный алгорим).

Каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем слепке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Неотслеживаемые файлы – это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что вы только взяли их из хранилища и ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, т.к. вы изменили их с момента последнего коммита. Вы индексируете эти изменения и затем фиксируете все индексированные изменения, а затем цикл повторяется.

Определение состояния файлов

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

git status

On branch master

nothing to commit

Это означает, что у вас чистый рабочий каталог, другими словами — в нём нет отслеживаемых изменённых файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. И наконец, команда сообщает вам на какой ветке (branch) вы сейчас находитесь. Пока что это всегда ветка master — это ветка по умолчанию.

Предположим, вы добавили в свой проект новый файл, простой файл README. Если этого файла раньше не было, и вы выполните git status, вы увидите свой неотслеживаемый файл вот так:

git status

On branch master

Untracked files:

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

README

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

Понять, что новый файл README неотслеживаемый можно по тому, что он находится в секции "Untracked files" в выводе команды status. Статус "неотслеживаемый файл", по сути, означает, что Git видит файл, отсутствующий в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить README, так давайте сделаем это.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

git add README

Если вы снова выполните команду `git status`, то увидите, что файл README теперь отслеживаемый и индексированный:

git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

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

Игнорирование файлов

Зачастую, имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т.п.). В таком случае, вы можете создать в корне проекта файл `.gitignore` с перечислением шаблонов соответствующих таким файлам. Вот пример файла .gitignore:

**.log*

.~

Первая строка предписывает Git'у игнорировать любые файлы заканчивающиеся на .log — файлы логов. Вторая строка предписывает игнорировать все файлы расширение которых начинается на тильду (~), такие расширения обычно используются для обозначения временных файлов. Вы можете также включить каталоги log, tmp. Хорошая практика заключается в настройке файла `.gitignore` до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите. К шаблонам в файле .gitignore применяются следующие правила:

** Пустые строки, а также строки, начинающиеся с #, игнорируются.*

** Можно использовать стандартные glob шаблоны.*

** Можно заканчивать шаблон символом слэша (/) для указания каталога.*

** Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.*

Glob-шаблоны представляют собой упрощённые регулярные выражения используемые командными интерпретаторами. Символ `*` соответствует 0 или более символам; последовательность [abc] — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; [0-9] соответствует любому символу из интервала (в данном случае от 0 до 9). Вот ещё один пример файла .gitignore:

```
# комментарий — эта строка игнорируется

# не обрабатывать файлы, имя которых заканчивается на .a

*.a

# НО отслеживать файл lib.a, несмотря на то, что мы игнорируем все. a файлы с помощью предыдущего правила

!lib.a

# игнорировать только файл TODO находящийся в корневом каталоге, не относится к файлам вида subdir/TODO

/TODO

# игнорировать все файлы в каталоге build/

build/

# игнорировать doc/notes.txt, но не doc/server/arch.txt

doc/*.txt

# игнорировать все .txt файлы в каталоге doc/

doc/**/*.txt
```

Просмотр индексированных и неиндексированных изменений

Если результат работы команды `git status` недостаточно информативен для вас – вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены – вы можете использовать команду `git diff`.

Подробно на этой команде останавливаться не будем, так как многие современные средства разработки (IDE - Интегрированная среда разработки) имеют встроенную поддержку команд *GIT* и изменения в файлах там наглядно отображаются.

Фиксация изменений

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

git commit

Эта команда откроет выбранный вами текстовый редактор. (Редактор устанавливается системной переменной $EDITOR, вы можете установить ваш любимый с помощью команды git config core.editor).

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

 # Please enter the commit message for your changes. Lines starting

 # with '#' will be ignored, and an empty message aborts the commit.

 # On branch master

 # Changes to be committed:

 #   (use "git reset HEAD <file>..." to unstage)

#

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

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

Можно сразу набрать свой комментарий к коммиту в командной строке вместе с командой commit, указав его после параметра -m:

```
git commit -m "мой первый коммит"
[master]: created 463dc4f: "мой первый коммит"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

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

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

Удаление файлов

Для того чтобы удалить файл из Git'a, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда git rm, которая также удаляет файл из вашего рабочего каталога, так что вы в следующий раз не увидите его как “неотслеживаемый”.

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции “Changes not staged for commit” (“Изменённые но не обновлённые” — читай не проиндексированные) вывода команды `git status`:

```
rm grit.gemspec
git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
    deleted:    grit.gemspec
```

Затем, если вы выполните команду `git rm`, удаление файла попадёт в индекс:

```
git rm grit.gemspec
rm 'grit.gemspec'
git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    deleted:    grit.gemspec
```

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра -f. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git'a.

Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на винчестере, и убрать его из-под бдительного ока Git'a. Это особенно полезно, если вы забыли добавить что-то в файл .gitignore и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию --cached:

git rm --cached readme.txt

В команду git rm можно передавать файлы, каталоги или glob-шаблоны. Это означает, что вы можете вытворять что-то вроде:

git rm log/.log*

Обратите внимание на обратный слэш (\\) перед *. Он необходим из-за того, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, которые имеют расширение .log в каталоге log/. Или же вы можете сделать вот так:

*git rm *~*

Эта команда удаляет все файлы, чьи имена заканчиваются на ~.

Просмотр истории коммитов

После того как вы создадите несколько коммитов, или же вы склонируете репозиторий с уже существующей историей коммитов, вы, при желании, можете узнать, что же происходило с этим репозиторием. Наиболее простой и в то же время мощный инструмент для этого — команда `git log`.

По умолчанию, `git log` выводит список коммитов созданных в данном репозитории в обратном хронологическом порядке. То есть самые последние коммиты показываются первыми.

Работа с удалёнными репозиториями

Чтобы иметь возможность совместной работы над каким-либо Git-проектом, необходимо знать, как управлять удалёнными репозиториями. Удалённые репозитории — это модификации проекта, которые хранятся в интернете или ещё где-то в сети. Их может быть несколько, каждый из которых, как правило, доступен для вас либо только на чтение, либо на чтение и запись. Совместная работа включает в себя управление удалёнными репозиториями и помещение (push) и получение (pull) данных в и из них тогда, когда нужно обменяться результатами работы.

Отображение удалённых репозиториев

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

Добавление удалённых репозиториев

Чтобы добавить новый удалённый Git-репозиторий под алиасом, к которому будет проще обращаться, выполните `git remote add [алиас] [url]`:

git remote add pb https://github.com/kolei/some_repo

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

Fetch и Pull

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

git fetch [имя удал. сервера]

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет.

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

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

Push

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

git push origin master

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

изменения и объединить с вашими. Только после этого вам будет позволено выполнить push.

Промежуточные итоги

К этому моменту вы умеете выполнять все базовые локальные операции с Git'ом: создавать или клонировать репозиторий, вносить изменения, индексировать и фиксировать эти изменения, а также просматривать историю всех изменений в репозитории. Дальше мы рассмотрим самую убийственную особенность Git'a – его модель ветвления.

Ветвление в Git

Почти каждая система контроля версий имеет в какой-то форме поддержку ветвления. Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию.

Пример ветвления из реального проекта:

Команда **git checkout [название ветки]**

Эта команда переключает ваш локальный репозитория на указанную ветку.

Тут нужно учитывать, что если в текущей ветке есть измененные файлы, то поменять ветку нельзя. Нужно либо зафиксировать изменения командами `git add` и `git commit`, либо отложить их командой `git stash`. Учитывая, что фиксировать нужно законченные действия, второй вариант бывает предпочтительнее.

Команда **git stash**

Команда `git stash` сохраняет незафиксированные изменения (подготовленные и неподготовленные) в отдельном хранилище, чтобы вы могли вернуться к ним позже. Затем происходит откат до исходной рабочей копии.

Теперь вы можете вносить изменения, создавать новые коммиты, переключаться между ветками и выполнять другие операции Git. По необходимости отложенные изменения можно будет применить позже.

Отложенные изменения сохраняются в локальном репозитории Git и не передаются на сервер при выполнении команды push.

Чтобы вернуть отложенные изменения нужно выполнить команду `git stash pop`.

При извлечении отложенных изменений они удаляются из набора и применяются к рабочей копии.

Вы также можете применить изменения к рабочей копии без удаления из набора отложенных изменений. Для этого воспользуйтесь командой `git stash apply`

Это полезно, если вам нужно применить одни и те же отложенные изменения к нескольким веткам.

Вы можете создать несколько наборов отложенных изменений. Команду `git stash` можно выполнить несколько раз, после чего у вас будет возможность просмотреть список наборов с помощью команды `git stash list`.

Рекомендуем добавлять к отложенным изменениям описание в качестве подсказки. Для этого используется команда `git stash save "сообщение"`

По умолчанию команда `git stash pop` применяет последний набор отложенных изменений: `stash@{0}`

Если вам нужно применить определенный набор ранее отложенных изменений, укажите его идентификатор в качестве последнего аргумента.

Создание новой ветки

Просто создать ветку можно командой `git branch название_ветки`.

После этого вы можете продолжить работу в текущей ветке или переключитсья на созданную командой `git checkout название_ветки`.

Можно сразу создать ветку и переключиться на неё командой `git checkout -b название_ветки`.

Завершение работы с веткой

Когда работа над новой фичей (в ветке) закончена, необходимо перенести изменния в основную ветку. Для этого используется команда `git merge название_ветки`. Т.е. Вы сначала переключаетесь на ту ветку, в которую хотите слить изменения, а затем объединяете их:

git checkout master

git merge название_ветки

Если кто-то менял те же файлы что и Вы (возможно даже Вы сами что-то меняли в другой ветке) и GIT не может автоматически определить куда вставить изменения, то возникнет ошибка слияния.

Для разрешения конфликта слияния вы должны либо выбрать один из вариантов (старая ветка или новая) или объединить оба варианта.

Формат MarkDown

В корне репозитория принято размещать файл readme.md, в котором описывается что это за репозиторий, для чего он нужен, инструкции по установке, если это необходимо.

Расширение `*.md` как раз и означает, что файл в формате MarkDown.

Markdown (произносится маркда́ун) — облегчённый язык разметки, созданный с целью обозначения форматирования в простом тексте, с максимальным сохранением его читаемости человеком, и пригодный для машинного преобразования (на GitHub-е как раз и отображается преобразованный текст). Раз это *простой текст*, то редактировать его можно даже в блокноте.

Текст с выделением

**выделение* (например, *курсив*)*

***сильное выделение** (например, **полужирное** начертание)*

*_**_**комбинация**_**_ курсива и полужирного начертания*

>Причем без разницы в каком порядке использовать символы ``*`` и ``_``

\~\~\~зачёркнутый~~\~\~ текст

Программный код

Выделяется знаком апострофа "\`", может быть как `внутри строки`

```txt
так
и
отдельным
блоком
```

Так можно оформлять не только программный код, но и любой текст, в котором нужно сохранить оригинальное форматирование. Дело в том, что MarkDown, как и практически любой язык верстки обрезает пустые строки и пробелы

Списки

1. 1\. Нумерованные

2. 2\. Причем порядок цифр

9. 9\. На результат не влияет

** обычные списки*

- с любым уровнем вложенности

1. можно комбинировать нумерованные и не нумерованные списки

Заголовки

Создание заголовков производится путём помещения знака решетки перед текстом заголовка. Количество знаков «#» соответствует уровню заголовка. Поддерживается 6 уровней заголовков.

 # \# Заголовок первого уровня

\#\#\# Заголовок третьего уровня

\#\#\#\#\#\# Заголовок шестого уровня

Цитаты (комментарии)

Любой элемент разметки (текст, список, картинка) могут быть помечены как цитата, если добавить в начале строки знак "\>"

Ссылки

** Ссылки на внешние ресурсы*

```
[Текст ссылки](http://example.com/ "Необязательный заголовок ссылки")
```

** Ссылки внутри документа*

```
[Текст ссылки](#якорь)
```

В качестве "якоря" может выступать [заголовок](#Заголовки) или html-тег \<a\> (но это уже нетривиальное использование разметки)

** Ссылки внутри репозитория*

```
[Текст ссылки](относительный путь на файл внутри репозитория)
```

Таблицы

столбец 1 | столбец 2

:---:|---:

1 | sfbvstb

2 | sbstbn

Как это выглядит в разметке (обратите внимание, содержимое столбцов можно центрировать или выравнивать по правому краю):

```
*столбец 1 | столбец 2*

*:---:|---:*

*1 | sfbvstb*

*2 | sbstbn*
```

Изображения

Если нужно разместить фотографию, скриншот или сложную схему, то можно добавить изображение

```
*![Alt-текст](http://example.com/ "Заголовок изображения")*
```

> Ссылка может быть и на изображение расположенное в репозитории

>```

>

Раздел №4. Модели применения GIT.

Существует множество способов применения git в проекте, а также сотрудничества с другими разработчиками. Помимо использования git для контроля версий кода вашего проекта, его можно использовать для координации того, как другие работают над проектом. Эффективнее всего этого можно достичь, применив какую-либо модель ветвления.

Workflow, то есть модель ветвления в git это практика, упрощающая работу над проектом. Её смысл в том чтобы организовать, добавление и публикацию изменений в исходном коде репозитория. Наиболее эффективным для проектов является наличие удаленного репозитория. Модель ветвления это не правило, которому нужно следовать, а скорее путеводитель. Поэтому можете просто выбрать определенные аспекты и сочетать их так как вам нужно.

Для этой статьи предположим, что ваша ветка это master и что вы знакомы с основными терминами git (ветка, коммит, слияние, pull request, patch, fork, репозиторий.

Здесь несколько моделей ветвления, которые вы можете рассмотреть для использования в своем проекте. Для каждой описаны суть работы и цель применения.

Central Workflow

Репозиторий содержит только одну главную ветку master. Все изменения комитятся в нее. Репозиторий может быть локальным, без удаленных копий или храниться удаленно, где он может быть клонирован или запушен.

Когда использовать

Идеально подходит для одиночного проекта. В своем проекте вы сможете использовать эту модель, для того чтобы видеть какие изменения произошли в течение процесса разработки. После проделанной работы вы коммитите изменения так, что позже сможете вернуться к любой из предыдущих версий. Также она отлично подойдет для небольших команд которые перешли от syn к git.

Developer Branch Workflow

У каждого разработчика есть своя личная ветка или несколько, в которые он пушит. Все изменения, опубликованные в удаленном репозитории будут в этой ветке. Вся работа может быть выполнена на разных ветках, но потом должна будет слита (merged) в одну главную ветвь.

Когда использовать

Больше подойдет для небольшого проекта с ограниченным количеством требований и небольшим количеством разработчиков, которым нужно чтобы их изменения в проекте были просмотрены до слияния с веткой master. Допустим у вас групповое задание, каждый участник делает свою часть, а затем публикует её в удаленном репозитории для того чтобы остальные её увидели до того, как она будет слита. В идеале это должно быть сделано с помощью запроса pull (merge). Также это может стать удобным способом для представления пулла в команде или организации.

Feature Branch Workflow

В своей простейшей форме репозиторий мог бы иметь основную ветку со стабильным, доступным кодом и другими ветками для разных фич (или багов, или улучшений), которые можно бы было интегрировать в главную ветку. То есть репозиторий будет иметь второстепенную основную ветку (dev) которая будет хранить тестируемый стабильный код для отправки пользователям, когда он будет слит с master. В этом случае ветка с фичами будет слита с dev, а не с master.

Когда использовать

Этот подход подойдет больше подходит командам, которые используют какой-то метод по управлению проектами (например Agile). Давайте скажем, что ваш проект находится в продолжительной разработке и вам нужно добавить набор фич до следующего релиза. Эти фичи назначены разным разработчикам, которые создают отдельную ветку для каждой опубликованной фичи до того, как она будет слита с dev для тестирования. Когда вы готовы к релизу, dev сливается с master.

Issue Branch Workflow

Очень похожа на предыдущую модель с одним лишь различием. Ветки создаются из заданий в проектном трекере. Ветки могут иметь одинаковые названия id заданий. И здесь только одна ветка на задание и одно задание на ветку.

Когда использовать

Лучше всего подойдет проектам, которые управляются по какому-то методу. Однако, несмотря на это, больше подходит тем проектам, в которых фичи готовятся не одним разработчиком. Например, если бы вы работали над самим интерфейсом, а другой разработчик бы работал над его другим аспектом. Он может быть применен в обоих проектах, релизы которых выходят постоянно или время от времени.

Forking Workflow

Благодаря этой модели, дополнения проекта осуществляются путем создания разветвления его репозитория. Все изменения фиксируются в любой ветке репозитория, а затем возвращаются в исходное хранилище с pull запросом. Разработчики будут иметь доступ только к чтению в удаленном репозитории.

Когда использовать

Чаще всего используется в проектах с открытым исходным кодом и публичными репозиториями. Каждый кто может просматривать репозиторий может сделать разветвление. До тех пор, пока они могут просматривать репозиторий им не нужен доступ для того чтобы внести изменения напрямую в репозиторий. Когда они закончили свою работу, они могут сделать запрос, который вы будете рассматривать и решите, что же с ним делать, интегрировать, отказать или просить доработать до окончательного слияния с проектом.

Patch Workflow

Используя этот подход, разработчики добавляют изменения в репозиторий вместе с патчем - файлом, который содержит все изменения в репозитории. Этот патч применяется кем-то, кто может напрямую писать в репозиторий, например maintainer/owner.

Когда использовать

Используется если разработчик не может напрямую писать в репозиторий, но имеет доступ к исходному коду. Если, например, вы поделились кодом своего проекта с другом или он получил доступ к вашему удаленному репозиторию. После тех изменений, которые вы закомиттили в их копию исходного кода создается патч и отправляется вам. Вы применяете их к репозиторию чтобы обновить его. Также используется теми, кто не является главным разработчиком проекта.

Вы можете найти еще миллион других способов взаимодействия в git или даже те, которые были описаны здесь, просто под другими названиями. Здесь только несколько моделей. Остальное остается на ваше усмотрение, выбрать одну из них или сочетать несколько. Просто выбирайте и пользуйтесь.

Модель GIT FLOW

Настройка репозитория, который мы используем, заключается в центральном «истинном» репозитории. Такая настройка отлично работает в данной модели. Обратите внимание, что этот репозиторий считается центральным (поскольку GIT является DVCS, такого понятия, как центральный репозиторий, на техническом уровне не существует). Мы будем называть этот репозиторий origin, так как это название знакомо всем пользователям GIT.

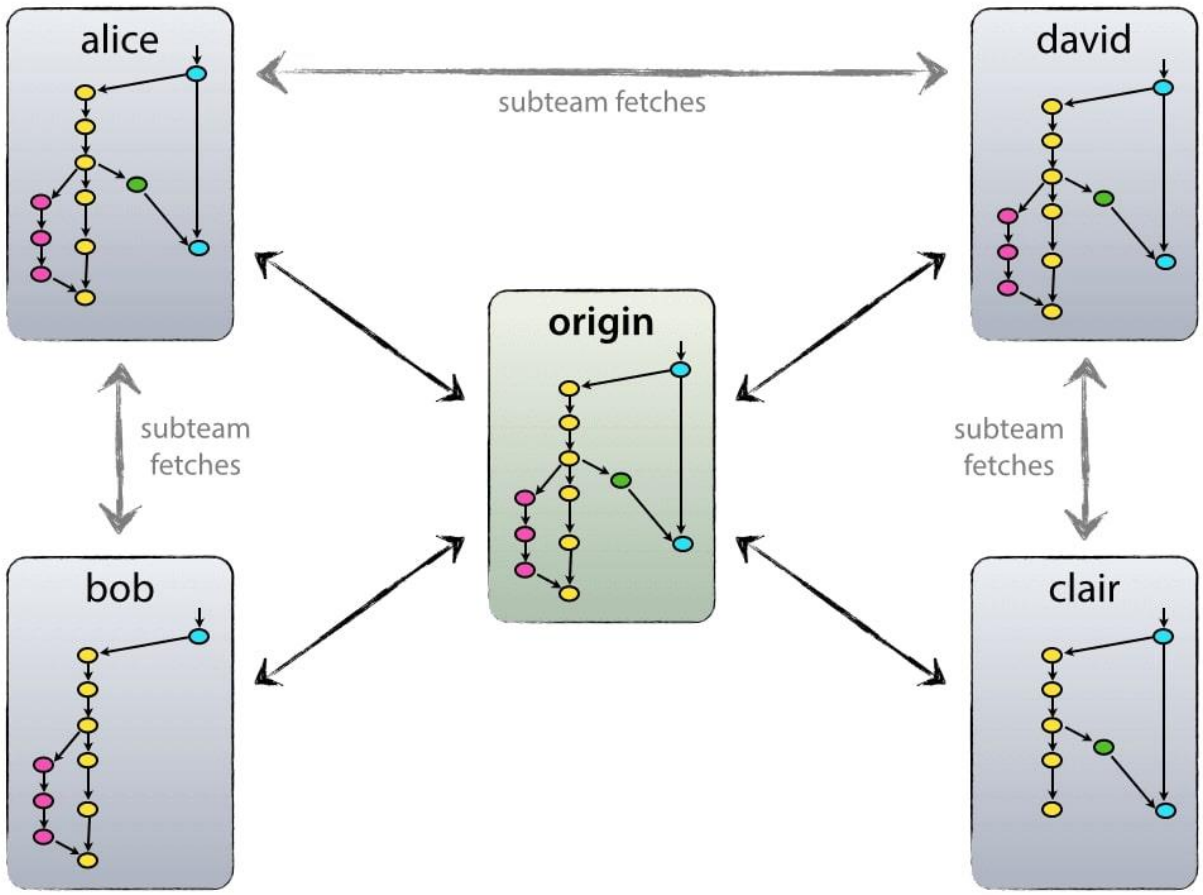


Рис. 25. Модель Git Flow

Каждый разработчик использует команды pull и push, обращаясь к origin. Помимо централизованных отношений pull-push, каждый разработчик может также извлекать изменения от других членов команды. Например, это может быть полезно для совместной работы с 2 или более разработчиками над большой новой функциональностью. Они не смогут отправить origin команду push преждевременно. На рисунке выше представлены подгруппы Алисы и Боба, Алисы и Дэвида, Клэр и Дэвида.

Технически это означает не что иное, как то, что Алиса определила удаленную ветку GIT с именем bob, указывая на хранилище Боба, и наоборот.

Основные ветки

По сути, данная модель разработки, связанная с GIT Flow, в значительной степени вдохновлена существующими моделями. Центральный репозиторий содержит две основные ветви: master и develop. Продолжительность их «жизни» бесконечна.

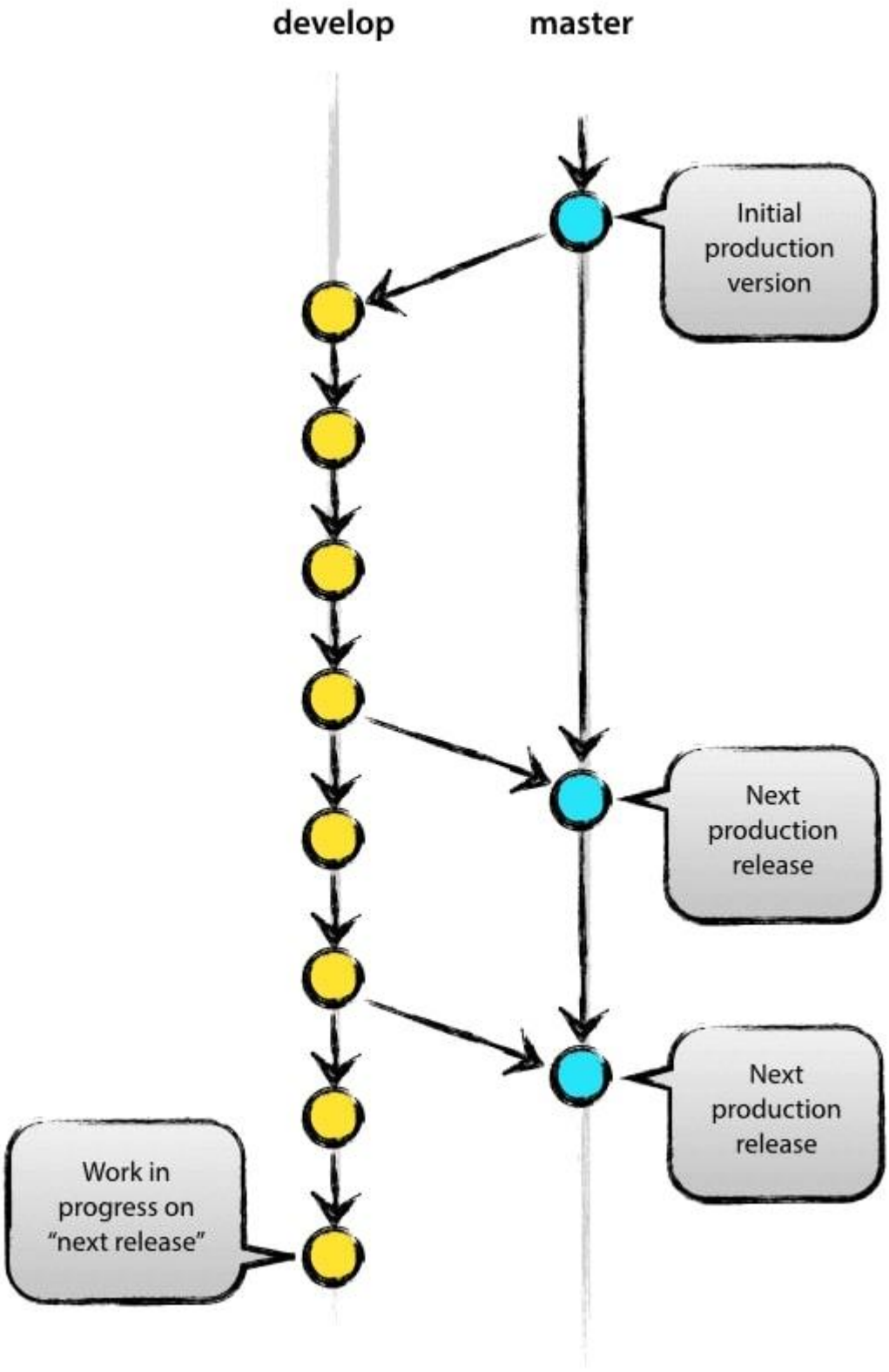


Рис. 26. Схема организации основных веток разработки

Ветка master в origin должна быть знакома каждому пользователю GIT. Параллельно с основной веткой существует другая ветвь, которая называется develop.

Мы считаем origin/master основной ветвью, где исходный код HEAD всегда отражает состояние на продакшене.

Мы считаем, что origin/develop является основной ветвью, где исходный код HEAD всегда отражает состояние с последними изменениями, которые могут потребоваться для разработки следующего релиза. Некоторые называют ее «интеграционной ветвью». Это то, из чего строятся любые автоматические сборки.

Когда исходный код в ветви develop достигает стабильной точки и готов к релизу, все изменения должны быть как-то объединены обратно в master, а затем помечены номером релиза. Подробнее этот момент будет обсуждаться ниже.

Следовательно, каждый раз, когда изменения объединяются с master, по определению появляется новый релиз. Мы, как правило, очень строго воспринимает это правило. Теоретически мы могли бы использовать скрипт GIT Hook для автоматической сборки и развертывания ПО на серверах каждый раз, когда выполняется коммит на master.

Второстепенные ветки, в которых ведется основная работа

Наряду с основными ветвями master и develop данная модель разработки, основанная на GIT Flow, использует множество второстепенных веток. В них в основном и ведется работа. Это способствует параллельной разработке между членами команды, облегчению отслеживания изменения функционала, подготовке к релизу рабочей версии. Это быстро решает возникающие проблемы. В отличие от основных ветвей, эти ветки всегда имеют ограниченный срок «жизни», так как в конечном итоге они будут удалены после завершения в них работы.

Мы можем использовать следующие типы веток:

- Feature branches (Ветки с новым/разрабатываем функционалом);
- Release branches (Ветки релизов);
- Hotfix branches (Ветки с быстрыми исправлениями).

Каждая из этих ветвей имеет определенную цель. Они связаны строгими правилами относительно того, из каких ветвей они выходят, а в какие должны слиться. Скоро мы это обсудим.

Эти ветви ни в коем случае не являются «особыми» с технической точки зрения. Типы ветвей классифицируются по тому, как мы их используем.

Feature branches

Могут ответвляться от:

develop

Должны вливаться в:

develop

Могут быть названы как угодно, кроме:

master, develop, release-, hotfix-**

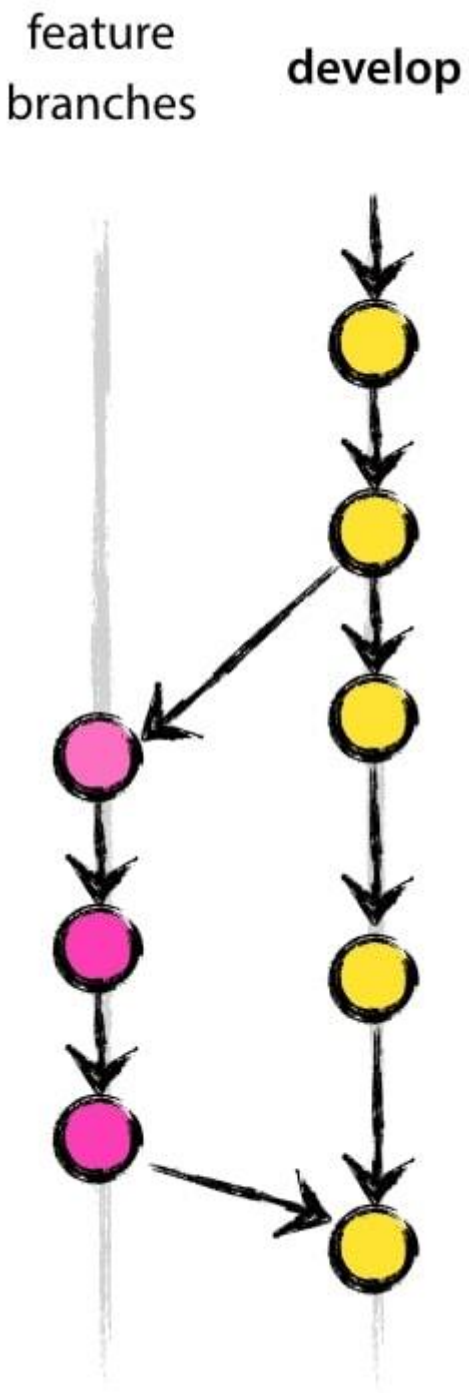


Рис. 27. Схема организации второстепенных веток разработки

Функциональные ветки используются для разработки нового функционала для предстоящего в скором или будущем релизе. При начале разработки функционала - целевой выпуск, в который будет включен этот код, может быть неизвестен в этот момент. Суть функциональной ветви заключается в том, что она существует до тех пор, пока код находится в стадии разработки. В конечном итоге ветка будет объединена с develop (для последующего релиза) или удалена (в случае неудачного эксперимента).

Эти ветви обычно существуют только в репозиториях разработчиков, а не в origin.

Когда разработчик начинает работу над новой задачей, ему необходимо сделать новую ветку. Она должна ответвляться от develop:

git checkout -b myfeature develop

Когда он закончит свою задачу, разработчику необходимо будет слить свою ветку в develop, что позволит в дальнейшем добавить его код в релиз на продакшен.

git checkout develop

git merge --no-ff myfeature

git branch -d myfeature

git push origin develop

Благодаря флагу --no-ff при слиянии всегда образуется новый коммит, даже если оно выполняется в режиме fast-forward. Это позволяет избежать потери информации о существовании ветки компонента и объединяет все коммиты, которые вместе добавляли новый функционал. Для сравнения:

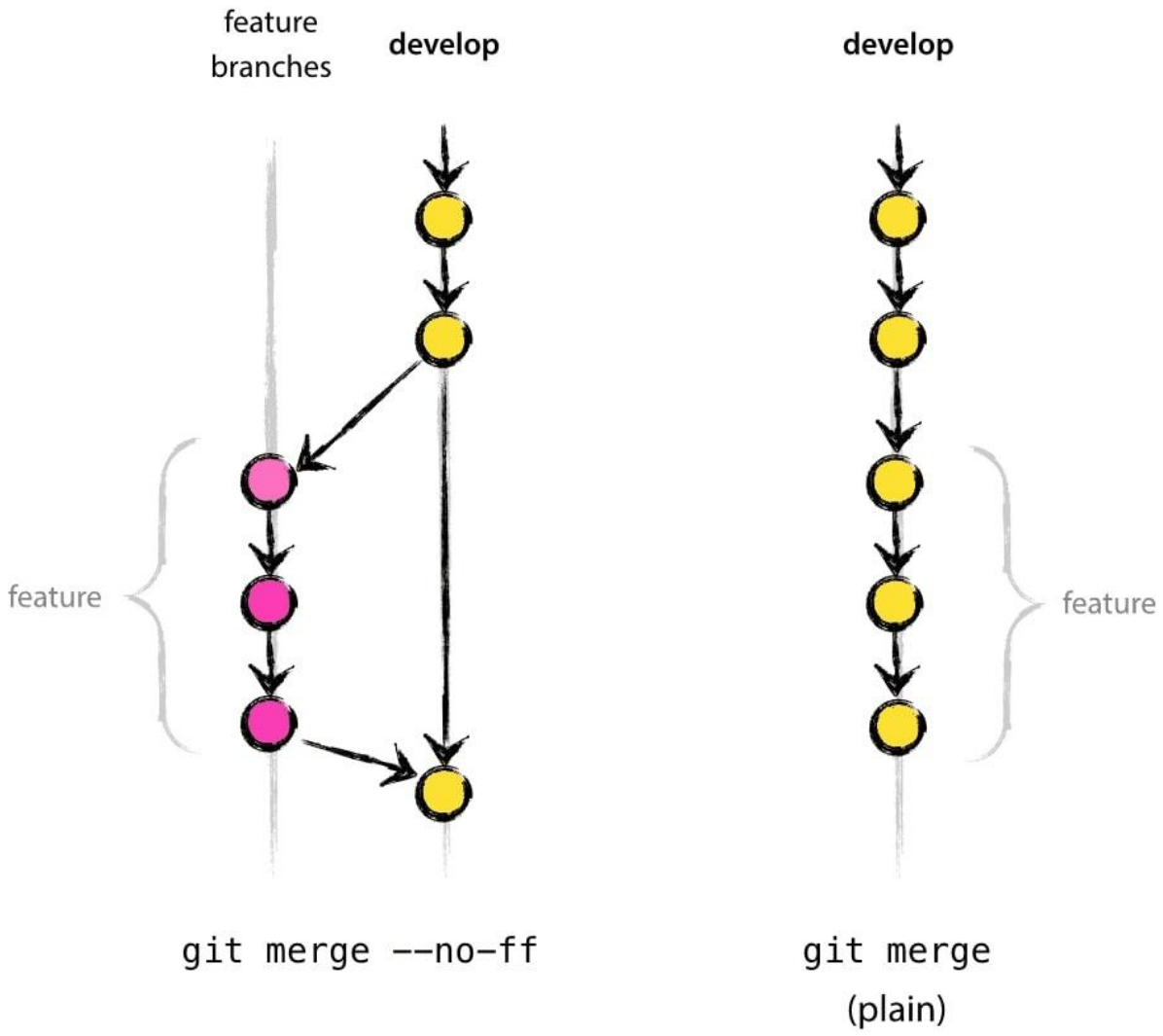


Рис. 28. Схема слияния веток разработки

В последнем случае в истории GIT невозможно увидеть, какие из коммитов вместе реализовали функционал. Вам придется вручную прочитать все сообщения log. Откатить все коммиты относящиеся к функционалу (т. е. группы коммитов) - настоящая головная боль в последней ситуации. Все было бы проще простого при использовании флага --no-ff.

Да, это создаст еще несколько (пустых) коммитов, но выигрыш намного больше, чем стоимость.

Release branches

Могут отходить от:

develop

Должны слиться с develop в:

master

Название:

*release-**

Ветки релизов необходимы для подготовки нового выпуска функционала продукта. В них обычно исправляют незначительные ошибки и подготавливают метаданные для релиза (номер версии, дата сборки и т. д.). Выполняя всю эту работу над веткой релиза, ветвь develop очищается для получения нового кода для следующего крупного релиза.

Ключевой момент, подходящий для отделения новой ветки выпуска от develop, - это то, когда разработка (почти) отражает желаемое состояние новой версии. На этот момент весь код, который должен быть выпущен, уже должен быть слит с develop. Все функции и исправления, нацеленные на будущие релизы, могут отсутствовать. Им придется подождать разветвлений ветви релиза.

Именно в начале ветки выпуска следующему релизу присваивается номер версии, не раньше. До этого момента ветвь develop отражала изменения для «следующего релиза». Вы не узнаете, станет ли этот «следующий релиз» 0.3 или 1.0, пока не будет запущена ветвь релиза. Это решение принимается в начале ветки релиза и выполняется в соответствии с правилами проекта по изменению номера версии.

Ветви выпусков создаются из ветки develop. Скажем, версия 1.1.5 является текущей версией, и у нас скоро будет большой выпуск. Состояние разработки готово к «следующему релизу», и мы решили, что он станет версией 1.2 (а не 1.1.6 или 2.0). Поэтому мы разветвляемся и даем ветке релиза имя, отражающее номер новой версии:

git checkout -b release-1.2 develop

./bump-version.sh 1.2

git commit -a -m "Bumped version number to 1.2"

После создания новой ветки и перехода на нее мы указываем номер версии. Здесь bump-version.sh представляет собой вымышленный сценарий оболочки, который изменяет некоторые файлы в рабочей копии, чтобы отразить новую версию. (Конечно, это можно делать вручную. Главное, чтобы поменялись некоторые файлы.) Затем фиксированный номер версии попадает в коммит.

Эта новая ветка может существовать некоторое время, пока релиз не будет точно выпущен. В течение этого времени исправления ошибок могут применяться в этой ветви (а не в ветви develop). Добавление большого нового функционала здесь строго запрещено. Ветки должны слиться с develop и, следовательно, ждать следующего крупного релиза.

Когда состояние ветки выпуска готово стать реальным релизом, необходимо выполнить некоторые действия. Во-первых, ветвь релиза объединяется с master (т. к. каждый коммит на master по определению является новым выпуском). Затем этот коммит на master должен быть помечен, чтобы потом было легко на него ссылаться. Наконец, изменения, внесенные в ветку

выпуска, необходимо объединить с develop, чтобы будущие выпуски также содержали эти исправления ошибок.

Первые 2 шага в GIT:

git checkout master

git merge --no-ff release-1.2

git tag -a 1.2

Релиз уже готов и помечен для дальнейшего использования.

Также можно использовать флажки -s или <key> для криптографической подписи тега.

Чтобы сохранить изменения, внесенные в ветку релиза, нужно слить их с develop. В GIT:

git checkout develop

git merge --no-ff release-1.2

Этот шаг вполне может привести к конфликту слияния. Это очень вероятно, так как мы сменили номер версии. Если такое произойдет, исправьте ошибку и создавайте коммит.

Теперь мы действительно закончили. Ветку релиза можно удалить, так как она нам больше не нужна:

git branch -d release-1.2

Hotfix branches

Могут отходить от:

master

Должны слиться с:

develop, master

Название:

*hotfix-**

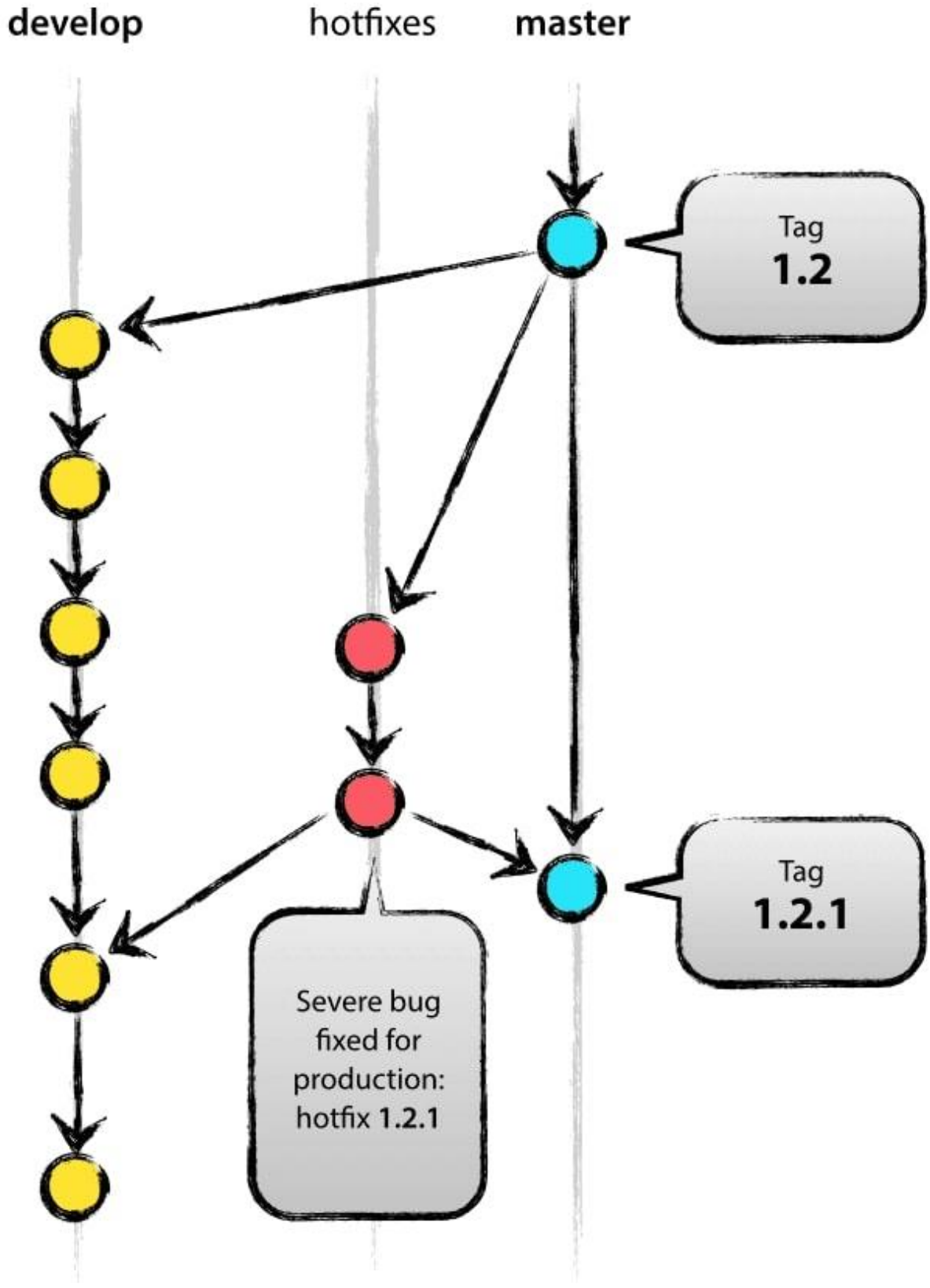


Рис. 29. Подготовка к релизу

Ветви исправлений очень похожи на ветки выпусков в том смысле, что они тоже предназначены для подготовки к новому релизу, пусть и незапланированному. Они возникают из-за необходимости действовать немедленно, когда состояние версии нежелательно. Когда критическая

ошибка в рабочей версии должна быть устранена немедленно, ветвь исправления может быть отделена от соответствующего тега в ветви master этой версии.

Суть в том, что работа членов команды (в ветви develop) может продолжаться, пока другой человек готовит быстрое исправление бага.

Ветки исправлений создаются из ветки master. Например, текущая рабочая версия 1.2. Она запущена, но в ней появляются проблемы из-за серьезной ошибки. Изменения в develop пока нестабильны. Тогда мы можем создать ветку исправлений и исправить баг:

git checkout -b hotfix-1.2.1 master

./bump-version.sh 1.2.1

git commit -a -m "Bumped version number to 1.2.1"

Не забывайте зафиксировать номер версии после создания ветки!

После исправления бага изменения фиксируются в 1 или нескольких коммитах:

git commit -m "Fixed severe production problem"

По окончании исправление необходимо слить с master. Также необходимо слить ветку и с develop, чтобы гарантировать, что исправление также будет включено в следующий выпуск.

Сначала обновите master и отметьте релиз:

git checkout master

git merge --no-ff hotfix-1.2.1

git tag -a 1.2.1

Также можно использовать флаги -s или <key> для криптографической подписи тега.

Далее включите исправление и в develop:

git checkout develop

git merge --no-ff hotfix-1.2.1

Единственное исключение из этого правила: если существует ветвь выпуска, исправления необходимо слить с ней, а не с develop. Обратное объединение исправления с веткой выпуска приведет к тому, что исправление также будет добавлено в develop, когда ветвь выпуска будет завершена. (Если работа в разработке требует немедленного исправления и не может ждать завершения ветки выпуска, можно смело вносить исправление в develop уже сейчас.)

Наконец, удалите временную ветку:

git branch -d hotfix-1.2.1

Заключение

В этой модели ветвления Git Flow нет ничего совершенно нового. «Общая картина», с которой начался этот пост, оказалась поразительно полезной в наших проектах. Она формирует великолепную ментальную модель, которую легко понять. Также между членами одной команды формируется взаимопонимание ветвлений и процессов выпусков.

Раздел №5. Заключение и выводы.

В данном разделе необходимо привести пронумерованный перечень выводов, которые были сделаны в результате установки и настройки системы Git и реализации модели применения Git.

Коллективное выполнение лабораторной работы

Этап I

За каждым из участников команды из трех человек должна быть зафиксирована роль, в соответствии с которой должны быть решены задачи, определенные в графе «Зона ответственности».

Роль	«Руководитель проектов в области информационных технологий»
Зона ответственности	Ход выполнения: – установить виртуальную машину в соответствии рекомендациями пункта «Среды виртуализации и виртуальные машины»; – выполнить простую установку программного продукта redmine; – авторизоваться в системе управления проектами и создать пользователя с правами администратора; – создать проект с названием согласно варианту задания; – создать список пользователей, соответствующий участникам команды; – разработать ролевую модель; – определить трекеры, статусы задач и последовательности действий; – в соответствии с определенной моделью работ в созданном проекте разработать дерево целей и задач на основе материалов первой лабораторной работы; – установить на виртуальной машине систему Git; – создать пользователя; – получить любой выбранный репозиторий средствами Git с информационного ресурса GitHub и загрузить в него условные файлы исходных кодов вместе с файлом описания README.md; – отразить последовательность выполнения шагов в итоговом отчете.
Приоритет при защите	первый

Роль	«Специалист по информационным системам»
Зона ответственности	Ход выполнения: – установить виртуальную машину в соответствии рекомендациями пункта «Среды виртуализации и виртуальные машины»; – загрузить пакет файлов с программным продуктом redmine на виртуальную машину; – выполнить детальную установку программного продукта redmine в соответствии с рекомендациями пункта «Установка и настройка систем управления проектами»; – выполнить установку темы оформления из комплекта поставляемых файлов; – установить на виртуальной машине систему Git; – создать пользователя; – получить любой выбранный репозиторий средствами Git с информационного ресурса GitHub и загрузить в него условные файлы исходных кодов вместе с файлом описания README.md; – отразить последовательность выполнения шагов в итоговом отчете.
Приоритет при защите	второй

Роль	«DevOps-инженер»
Зона ответственности	Ход выполнения: – установить виртуальную машину в соответствии рекомендациями пункта «Среды виртуализации и виртуальные машины»; – установить на виртуальной машине систему Git; – создать 4-х пользователей; – создать репозиторий и загрузить условные файлы исходных кодов вместе с файлом описания README.md; – реализовать модель Git Flow в рамках созданного репозитория; – отразить варианты использования модели Git Flow в итоговом отчете.
Приоритет при защите	третий

Подготовка к защите лабораторной работы

Каждый участник команды из трех человек должен по результатам проведенной работы подготовить устный доклад на 3-5 минут, в рамках которого он должен пояснить, что было сделано, какие возникли сложности, проблемы в реализации этапов и какие были сделаны выводы в процессе их реализации.

Руководитель проектов в области информационных технологий должен быть готов ответить на следующие вопросы.

1. Какие атрибуты являются ключевыми для формирования модели работ по проекту?
2. От каких параметров модели работ зависит жизненный цикл задачи?
3. Опишите ролевую модель, разработанную под вариант.

Специалист по информационным системам должен быть готов ответить на следующие вопросы.

1. На каких технологиях разработана система управления проектами на базе продукта Redmine?
2. С какими СУБД Redmine поддерживает интеграцию?
3. Опишите схему web-приложения Redmine.

DevOps-инженер должен быть готов ответить на следующие вопросы.

1. К какому виду систем контроля версий относится система контроля версий Git?
2. Какие модели применения системы контроля версий Git Вам известны?
3. Опишите модель Git Flow.

Список литературы

1. Чакон С., Штрауб Б. Git для профессионального программиста. - СПб.: Питер, 2016. – 496 с.: ил. – (Серия «Библиотека программиста»). ISBN 978-5-496-01763-3.
2. Система управления проектами Redmine. [Электронный ресурс]. URL: https://www.redmine.org/ (дата обращения 10.07.2021).

Варианты заданий для выполнения лабораторных работ

Номер варианта задания	Задание на лабораторную работу
1	Разработка программного обеспечения для автоматизированной/информационной системы железной дороги
2	Разработка программного обеспечения для автоматизированной/информационной системы авиакомпании
3	Разработка программного обеспечения для автоматизированной/информационной системы аэропорта
4	Разработка программного обеспечения для автоматизированной/информационной системы морского порта
5	Разработка программного обеспечения для автоматизированной/информационной системы автобусного вокзала
6	Разработка программного обеспечения для автоматизированной/информационной системы школы
7	Разработка программного обеспечения для автоматизированной/информационной системы библиотеки
8	Разработка программного обеспечения для автоматизированной/информационной системы университета
9	Разработка программного обеспечения для автоматизированной/информационной системы службы занятости
10	Разработка программного обеспечения для автоматизированной/информационной системы службы социальной защиты
11	Разработка программного обеспечения для автоматизированной/информационной системы поликлиники
12	Разработка программного обеспечения для автоматизированной/информационной системы обязательного медицинского страхования
13	Разработка программного обеспечения для автоматизированной/информационной системы пенсионного фонда
14	Разработка программного обеспечения для автоматизированной/информационной системы выставочного комплекса
15	Разработка программного обеспечения для автоматизированной/информационной системы для организации НИОКР

16	Разработка программного обеспечения для автоматизированной/информационной системы издательства
17	Разработка программного обеспечения для автоматизированной/информационной системы редакции газеты
18	Разработка программного обеспечения для автоматизированной/информационной системы типографии
19	Разработка программного обеспечения для автоматизированной/информационной системы гостиницы
20	Разработка программного обеспечения для автоматизированной/информационной системы киноцентра
21	Разработка программного обеспечения для автоматизированной/информационной системы фирмы по прокату автомобилей
22	Разработка программного обеспечения для автоматизированной/информационной системы букмекерской фирмы
23	Разработка программного обеспечения для автоматизированной/информационной системы фондовой биржи
24	Разработка программного обеспечения для автоматизированной/информационной системы банка
25	Разработка программного обеспечения для автоматизированной/информационной системы лизинговой компании
26	Разработка программного обеспечения для автоматизированной/информационной системы туристического агентства
27	Разработка программного обеспечения для автоматизированной/информационной системы фильмотеки
28	Разработка программного обеспечения для автоматизированной/информационной системы агентства недвижимости
29	Разработка программного обеспечения для автоматизированной/информационной системы страховой организации
30	Разработка программного обеспечения для автоматизированной/информационной системы автошколы
31	Разработка программного обеспечения для автоматизированной/информационной системы оператора связи
32	Разработка программного обеспечения для автоматизированной/информационной системы автосервиса
33	Разработка программного обеспечения для автоматизированной/информационной системы для оказания госуслуг

34	Разработка программного обеспечения для автоматизированной/информационной системы фирмы по сборке и продаже компьютеров и комплектующих
35	Разработка программного обеспечения для автоматизированной/информационной системы транспортной фирмы
36	Разработка программного обеспечения для автоматизированной/информационной системы супермаркета
37	Разработка программного обеспечения для автоматизированной/информационной системы книжного магазина
38	Разработка программного обеспечения для автоматизированной/информационной системы ломбарда
39	Разработка программного обеспечения для автоматизированной/информационной системы ГИБДД
40	Разработка программного обеспечения для автоматизированной/информационной системы спортивного клуба
41	Разработка программного обеспечения для автоматизированной/информационной системы интернет-провайдера
42	Разработка программного обеспечения для автоматизированной/информационной системы интернет-магазина
43	Разработка программного обеспечения для автоматизированной/информационной системы интернет-аукциона
44	Разработка программного обеспечения для автоматизированной/информационной системы почтовой службы
45	Разработка программного обеспечения для автоматизированной/информационной системы предприятия ЖКХ
46	Разработка программного обеспечения для автоматизированной/информационной системы рекламного агентства
47	Разработка программного обеспечения для автоматизированной/информационной системы курьерской фирмы
48	Разработка программного обеспечения для автоматизированной/информационной системы ресторанного комплекса
49	Разработка программного обеспечения для автоматизированной/информационной системы службы такси
50	Разработка программного обеспечения для автоматизированной/информационной системы службы технической поддержки
51	Разработка программного обеспечения для автоматизированной/информационной системы <свой вариант> (необходимо сформулировать тему)