

Программирование

Система управления версиями git

Кафедра ИВТ и ПМ

2018

Трудности разработки

Во время разработки требуется хранить как минимум две версии исходного кода программы: версию находящийся в активной разработке и последнюю гарантировано рабочую версию.

Вторая версия нужна, чтобы в любой момент можно было откатиться к рабочему коду, провести дополнительные тесты не прерывая разработку или продемонстрировать заказчику.

Трудности разработки

Создание таких резервных копий рабочего вручную вызывает дополнительные трудности.

Полное копирование может занимать время и отвлекать от работы. При частичном копировании нужно самостоятельно следить за тем, какие именно файлы изменялись.

За созданными копиями приходится следить, удалять старые версии, контролировать даты создания.

Без дополнительных средств сравнение текущей версии и работающей копии проблематично. А это может понадобится, чтобы отследить изменения, которые привели к неработоспособности программы.

Трудности разработки

Если возникает необходимость внесения текущих изменений и исправить, например, критический баг то может возникнуть уже две версии программы находящиеся в разработке. Которые потом потребуется объединить.

Кроме того, совместная разработка одного проекта приведёт к необходимости согласовывать изменения, вручную вносить изменения в копию проекта каждого разработчика.

Решить эту проблемы, автоматизировать рутинные операции призвана **система управления версиями**.

Система управления версий

Система управления версиями (Version Control System, VCS)

— программное обеспечение для облегчения работы с изменяющейся информацией.

Позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Репозиторий, хранилище — место, где хранятся и поддерживаются какие-либо данные.

Outline

Локальный репозиторий

Удалённый репозиторий

Графические оболочки и интеграция

Начало работы

1. создать репозиторий

```
git init
```

2. Добавить файлы к отслеживанию

```
git add файлы
```

Как правило в отслеживании должны быть файлы исходных кодов и другие файлы, необходимые для компиляции и запуска программы. Исполняемые файлы не отслеживаются. Потому, что их всегда можно получить после компиляции и чтобы не засорять ими репозиторий.

3. Просмотреть список отслеживаемых файлов.

```
git ls-files
```


Фиксация изменений

Git не запоминает изменения в реальном времени.

Это происходит потому, что каждое изменение должно быть логически завершённым. А это решает разработчик.

Чтобы "сделать фотографию" или записать текущее состояние файлов используется команда **commit**.

Такое действие называется **фиксацией** или **коммитом** (commit)

Фиксация изменений

Зафиксировать изменения (сделать коммит)

```
git commit -am "кратко об изменениях"
```

Ключи команды commit:

-a - добавить все отслеживаемые файлы в фиксацию

-m - комментарий к фиксации

В комментариях следует кратко описывать сделанные изменения.

Например: „добавлена функция генерации врагов“ или „исправлен баг с отрисовкой героя“.

Так как описание коммита должно быть кратким, то используют сокращения и условные обозначения для частых действий.

Описание коммитов

	КОММЕНТАРИЙ	ДАТА
○	НАПИСАЛ ГЛАВНЫЙ ЦИКЛ И УПРАВЛЕНИЕ ТАЙМЕРОМ	14 ЧАСОВ НАЗАД
○	ДОБАВИЛ ПАРСИНГ ФАЙЛА НАСТРОЕК	9 ЧАСОВ НАЗАД
○	РАЗНЫЕ БАГФИКСЫ	5 ЧАСОВ НАЗАД
○	ТАМ ДОБАВИЛ, ТУТ ИСПРАВИЛ	4 ЧАСА НАЗАД
○	БОЛЬШЕ КОДА	4 ЧАСА НАЗАД
○	ВОТ ТЕБЕ ЕЩЁ КОД	4 ЧАСА НАЗАД
○	АААААААА	3 ЧАСА НАЗАД
○	ФВЛАОЫДЛВАОЫВЛДАО	3 ЧАСА НАЗАД
○	МОИ ПАЛЬЦЫ НАБИРАЮТ СЛОВА	2 ЧАСА НАЗАД
○	ПАААААЛЪЦЫЫЫЫЫ	2 ЧАСА НАЗАД

ЧЕМ Дольше тянется проект, тем менее информативны
сообщения моих git-коммитов.

В комментариях следует кратко описывать сделанные изменения.

Например: „добавлена функция foo()“ или „исправлен баг с
отрисовкой героя“.

Типичный сценарий использования

Небольшие изменения.

1. Внести изменения.
2. Протестировать.

При необходимости просмотреть изменения:

```
git diff
```

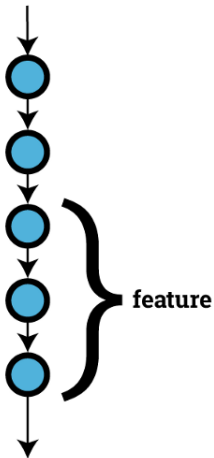
3. Зафиксировать изменения (сделать коммит)

```
git commit -am "кратко об изменениях"
```

Типичный сценарий использования

Последовательность внесённых изменений

master



Внесение изменений

Фиксируемые изменения должны быть логически завершёнными.

Это означает, что после внесения изменений программа должна быть синтаксически правильной и работать корректно.

Нужно рассматривать комиты (внесение и фиксация изменений) как неделимые, атомарные действия в разработке программы.

Внесение изменений

Если планируются обширные изменения, то стоит подумать над созданием отдельной ветви, чтобы параллельно существовала исходная версия программы и версия, в которую вносятся изменения - *рабочая версия*.

Допускается, что рабочая версия может не транслироваться, работать с ошибками.

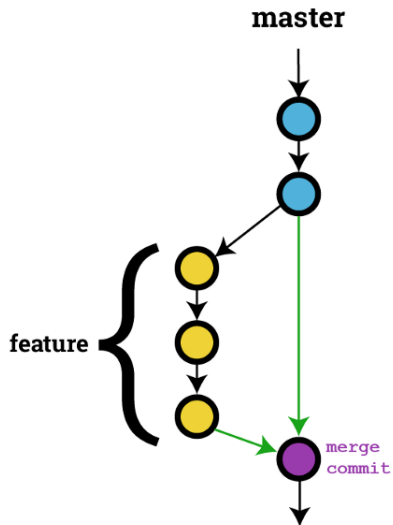
Однако в любой момент должна быть возможность вернуться к исправной версии программы.

Внесение изменений

После того как изменения будут закончены, программа станет синтаксически правильной и протестирована, изменения рабочей версии добавляются к основной версии.

При внесении новых изменений, снова создаётся рабочая версия и всё повторяется заново.

Внесение изменений



Внесение изменений

В git отдельные версии программы хранятся в **ветках** (branches).

Одновременно с созданием репозитория создаётся основная ветка - **master**

Команды для работы с ветками

branch <имя_ветки> - создание ветки

checkout <имя_ветки> - переключение на ветку

checkout -b <имя_ветки> - создание ветки и переключение на неё

merge <имя_ветки> - объединение текущей ветки с другой

Переключится с ветки на ветку можно только если в текущей ветке все изменения зафиксированы.

Типичный сценарий использования

Значительные изменения

- ▶ Создать рабочую ветку и переключится
`git checkout -b new_feature`
- ▶ Внести изменения
 - ▶ `commit 1`
 - ▶ `commit 2`
 - ▶
 - ▶ `commit n`
- ▶ Переключится на основную ветку
`git checkout master`
- ▶ Объединить основную ветку с рабочей
`git merge new_feature`

git worktree. Ветки в отдельных каталогах

Git позволяет организовать одновременный доступ к разным веткам, сохранив их в отдельных каталогах.

Сохранении ветки `my_another_branch` в каталог `myprog_another_branch`.

```
git worktree add ../myprog_another_branch my_another_branch
```

Как работать с Git worktree: краткая инструкция

Outline

Локальный репозиторий

Удалённый репозиторий

Графические оболочки и интеграция

Удалённый репозиторий

Локальный репозиторий позволяет легко отслеживать изменения, хранить несколько версий программы. Всегда есть возможность вернуться к предыдущей версии.

Удалённый репозиторий




Удалённый репозиторий помимо этого делает удобной групповую разработку: разработчик отправляет свои изменения в общий удалённый репозиторий и забирают из него изменения сделанные другими разработчиками.

Удалённый репозиторий

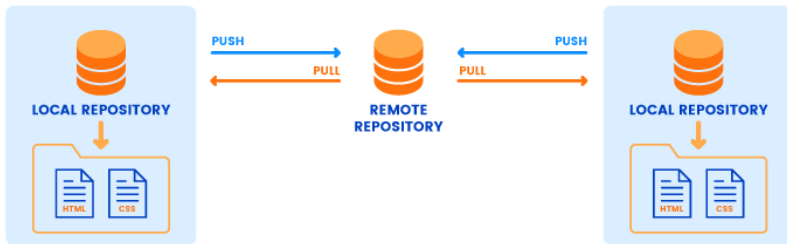
Кроме того, удалённый репозиторий можно рассматривать как резервную копию локального.

In case of fire



-  1. **git commit**
-  2. **git push**
-  3. **leave building**

Удалённый и локальные репозитории



Удалённый репозиторий

Создание удалённого репозитория на основе локального

1. Создать удалённый репозиторий на сайте (например github). Запомнить адрес репозитория.
2. Настройка локального репозитория.

Он должен знать об удалённом:

```
git remote add origin git@github.com:Username/Reponame.g
```

3. Отправка ветки master в удалённый репозиторий

```
git push [удал. сервер] [ветка]
```

```
git push -u origin master
```

origin - псевдоним для удалённого репозитория.

Удалённый репозиторий

Создание локальной копии удалённого репозитория.

```
git clone git://github.com/Username/Reponame.git
```

Веб хостинги использующие git

 Bitbucket

GitHub



GitLab

Outline

Локальный репозиторий

Удалённый репозиторий

Графические оболочки и интеграция

Графические оболочки

Список графических оболочек для Git

Интеграция git и PyCharm

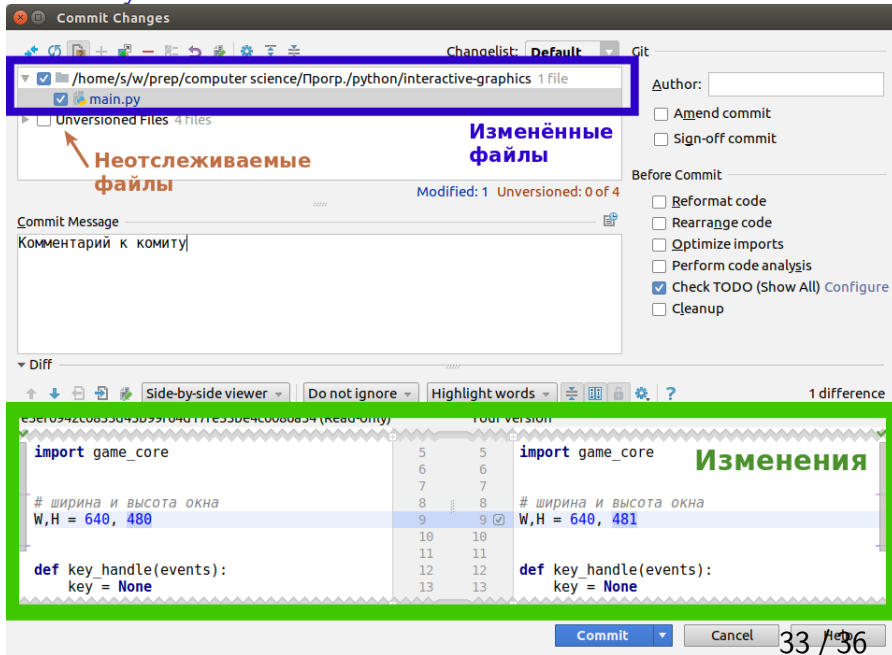
PyCharm автоматически определяет, создан ли локальный репозиторий для данного проекта (каталога).

Если репозиторий не создан,

Комит в PyCharm

- ▶ Меню VCS -> commit
или
- ▶ Crtr + K
или
- ▶ кнопка commit на панели инструментов.

Комит в PyCharm



The image shows the PyCharm 'Commit Changes' dialog and a diff view. The dialog is titled 'Commit Changes' and shows the file path '/home/s/w/rep/computer science/Прорп./python/interactive-graphics 1 file' and the file 'main.py'. A blue box highlights the file list. Below the file list, there is a section for 'Unversioned Files' with a red arrow pointing to it and the text 'Неотслеживаемые файлы' (Untracked files). The 'Commit Message' field contains 'Комментарий к комиту'. On the right, there are options for 'Author', 'Amend commit', 'Sign-off commit', and 'Before Commit' actions. The diff view at the bottom shows a comparison between the 'Read-only' version and the 'Your version'. The diff view highlights a change in the 'W,H' variable from 480 to 481. A green box highlights the diff view. The word 'Изменения' (Changes) is written in green next to the diff view.

Commit Changes

Changelist: Default

Git

Author:

☐ Amend commit

☐ Sign-off commit

Before Commit

☐ Reformat code

☐ Rearrange code

☐ Optimize imports

☐ Perform code analysis

☒ Check TODO (Show All) [Configure](#)

☐ Cleanup

Commit Message

Комментарий к комиту

Modified: 1 Unversioned: 0 of 4

Diff

Side-by-side viewer Do not ignore Highlight words

1 difference

es1e0942c0835d43b991b4b171e33be4c0080834 (Read-only)

Your version

import game_core

ширина и высота окна

W,H = 640, 480

def key_handle(events):

key = None

import game_core

ширина и высота окна

W,H = 640, 481

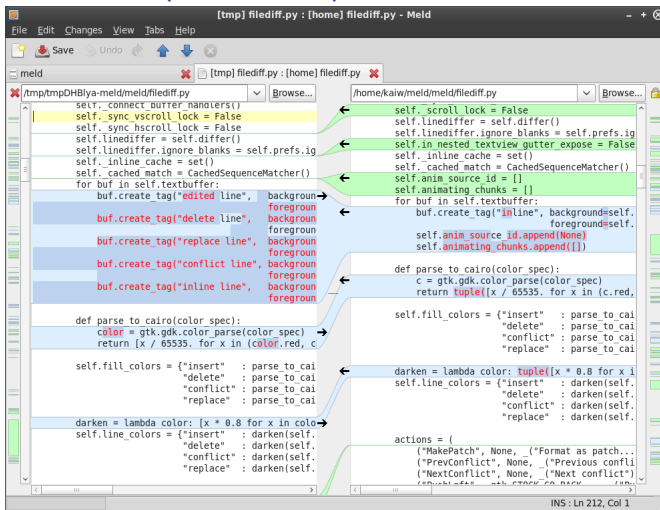
def key_handle(events):

key = None

Изменения

Commit Cancel

Дополнительно. Сравнение файлов



Meld - программа для построчного сравнения¹ двух файлов.

¹см. также ru.wikipedia.org/wiki/Diff

Ссылки и литература

- ▶ git-scm.com/book/ru/v2 - документация git
- ▶ youtube: GitHub быстрый старт в PyCharm
- ▶ Список графических оболочек для Git

Ссылки и литература

Ссылка на слайды

github.com/VetrovSV/Programming