Урок Репозитории 2

Совместная работа над проектом, основные понятия и команды. Работа с репозиториями в среде разработки

Аннотация

В этом уроке рассматривается работа с ветками в Git, решение конфликтов изменений, а также основы командной разработки в Git.

1. Ветки в Git

Мы уже немного затронули тему о **ветках** в Git. Настало время внимательней разобрать это понятие.

Если бы система контроля версий ограничивалась только ведением истории коммитов, то она бы никак не помогла нескольким разработчикам работать с одним репозиторием. Или одному разработчику — над несколькими задачами в одном.

Для всего этого (и не только) в Git и существуют ветки. Ветки позволяют:

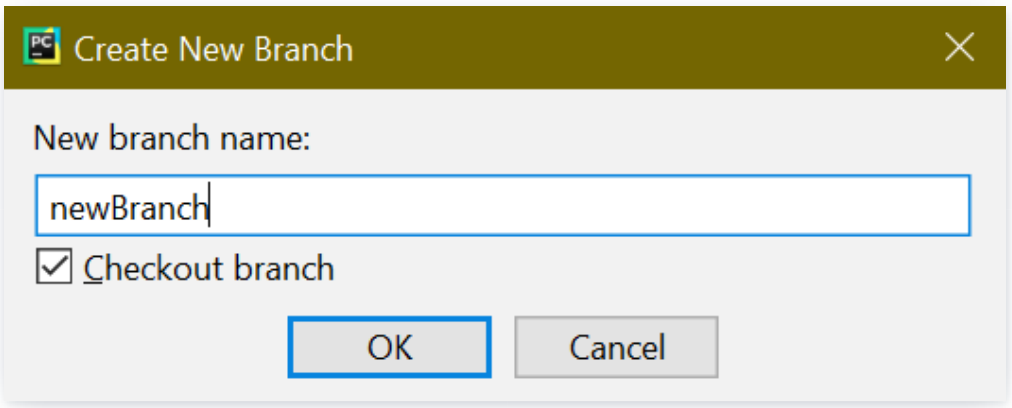
- Давать имена версиям
- Иметь одновременно несколько **рабочих** версий (рабочей версией называется та, над которой в определенный момент времени трудится разработчик)
- Объединять результаты деятельности нескольких разработчиков

Допустим, у нас есть программа (можно продолжить работу с репозиторием из прошлого урока, а можно создать новый репозиторий), последняя закоммиченная версия которой выглядит так:

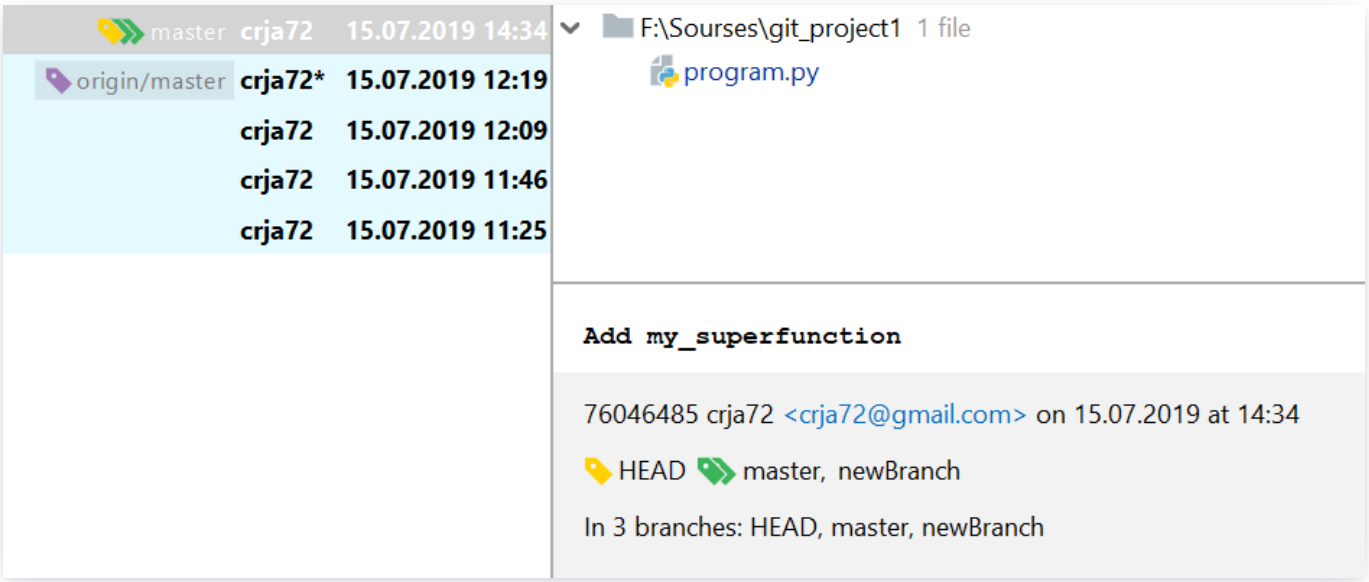
```
def my_superfunction():
    print('What an awesome print!')


def main():
    print('My first git program')
    print('And I change it every day')
    print('Again')
    print('UFO came and added this line')
    my_superfunction()


if __name__ == '__main__':
    main()
```

Мы хотим создать версии нашей программы, в которой функция `my_superfunction` ведет себя по-другому, например, одна выводит надпись Hello, python!!!, другая — Yandexlyceum, но старую версию мы тоже хотим сохранить.

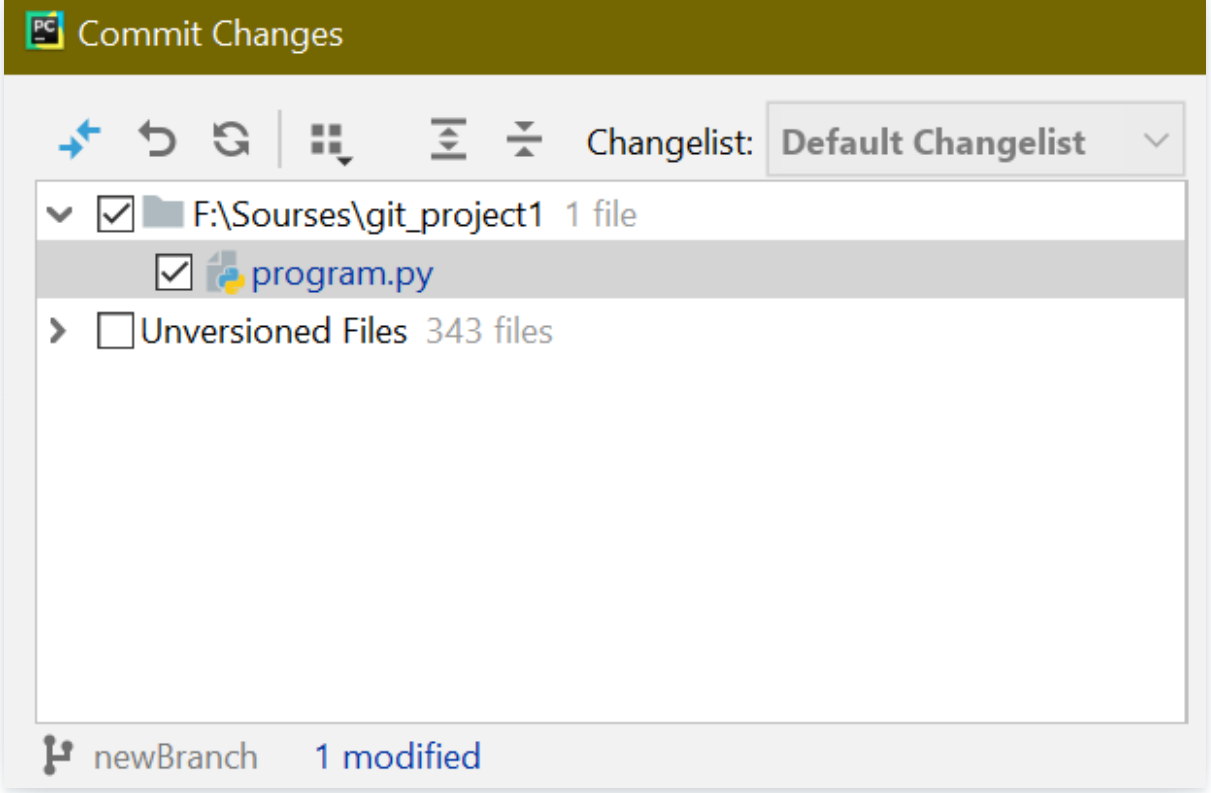
Сначала заведем новую ветку программы и сразу на нее переключимся. Это можно сделать через пункт меню VCS → Git → Branches → New Branch (или Ctrl + Shift + `). В появившемся окне введем имя нашей новой ветки, например, `newBranch`, оставим галочку, которая сразу осуществляет переключение на новою ветку при создании нетронутой.



Посмотрим на вкладку Log меню системы контроля версий. Обратите внимание, что на последнем коммите появилась еще одна зеленая бирочка, которая говорит о том, что мы находимся на развилке и в этом месте код двух веток master и newBranch совпадает.



Поменяем код нашей функции и закоммитим изменения.

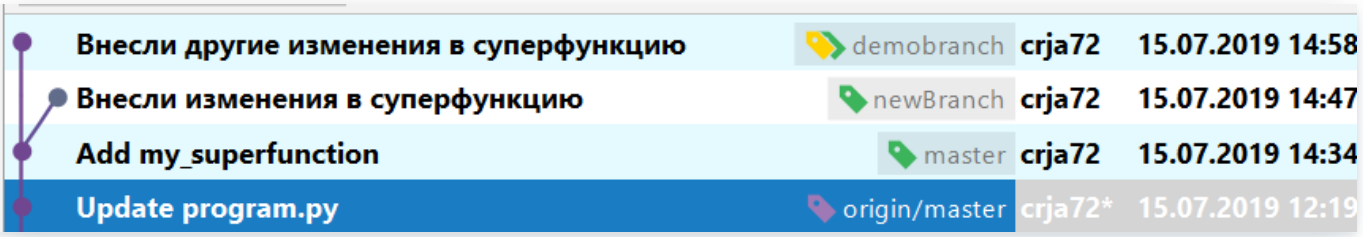


При совершении коммита видно, что изменения фиксируются в ветку newBranch.

Давайте вернемся в master. В Logs в меню системы контроля версии выберем последний коммит ветки master, кликнем на нем правой кнопкой мышки и выберем Branch 'master' → Checkout.

Создадим еще одну ветку, назовем ее demobranch.

Внесем изменения в функцию `my_superfunction`, чтобы она выводила текст Yandexlyceum, и закомитим изменения.



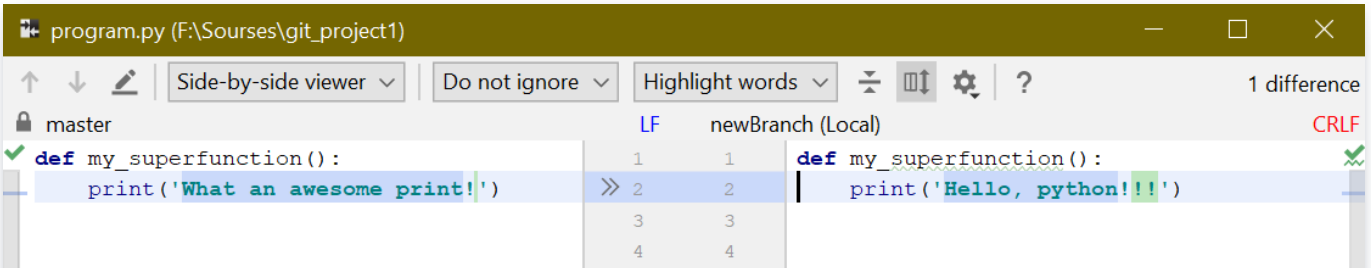
По логам видно, что у нас есть три ветки: master, newBranch и demobranch. demobranch — текущая активная ветка, то, что мы находимся в ней, подтверждается указателем HEAD. Также есть ветка newBranch, у которой с demobranch общий источник кода — последний на текущий момент коммит ветки master.

Для того чтобы переключиться в нужную ветку, надо кликнуть на последнем коммите этой ветки и выбрать пункт меню Branch 'имя_ветки' → Checkout (это можно сделать и через меню: VCS → Git → Branches).

Теперь удалим неиспользуемую ветку demobranch командой VCS → Git → Branches → demobranch → Delete.

Теперь мы умеем создавать и удалять ветки, а также переключаться между ветками.

Версии программы, зафиксированные в разных ветках, можно сравнивать между собой с помощью команды VCS → Git → Compare with Branch. Давайте сравним program.py из ветки newBranch, в которой мы сейчас находимся, и master.



```
def main():
    print('My first git program')
    print('And I change it every day')
    print('Again')
    print('UFO came and added this line'
    my_superfunction()


if __name__ == '__main__':
    main()
```

```
def main():
    print('My first git program')
    print('And I change it every day')
    print('Again')
    print('UFO came and added this line')
    my_superfunction()


if __name__ == '__main__':
    main()
```

Обратите внимание: создавая ветку, мы **получаем полную** копию ветки, в которой находились в этот момент.

Для закрепления успехов перейдем на ветку master, создадим новую ветку addAuthorBranch, и добавим в начало нашей программы комментарий с именем автора:

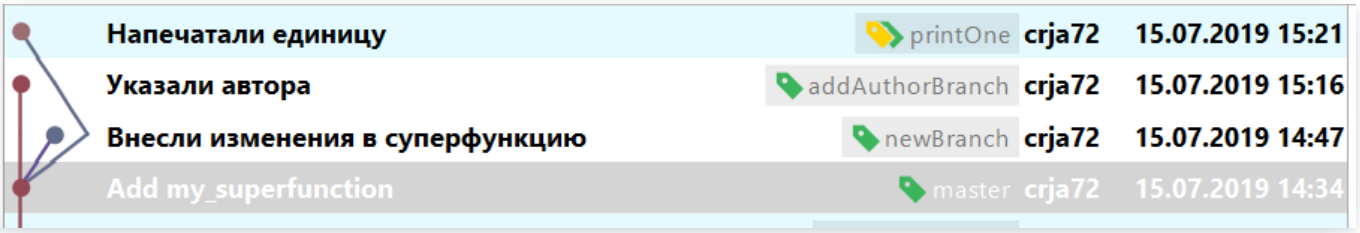
```
# Yandexlyceum
def my_superfunction():
    print('What an awesome print!')
....
```

Закоммитим изменения.

Затем снова вернемся в master, создадим ветку printOne, в которой добавим в конец функции `main` код:

```
print(1)
```

Снова закоммитим изменения. После проделанной работы Log системы контроля версий должен выглядеть примерно вот так:

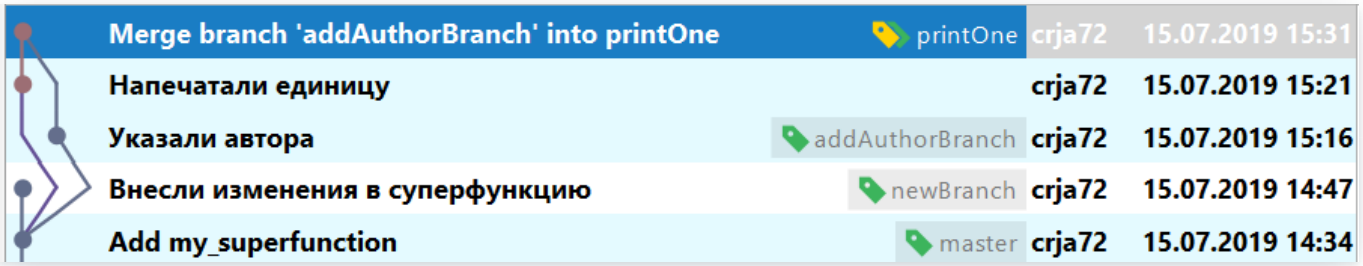


2. Объединение (слияние) изменений

Итак, Git позволяет независимо разрабатывать несколько версий программы в разных **ветках** одного большого дерева.

Теперь разберемся, как из двух веток собрать единую версию. Попробуем получить программу как с указанием авторства из ветки addAuthorBranch, так и с печатью единицы из ветки printOne.

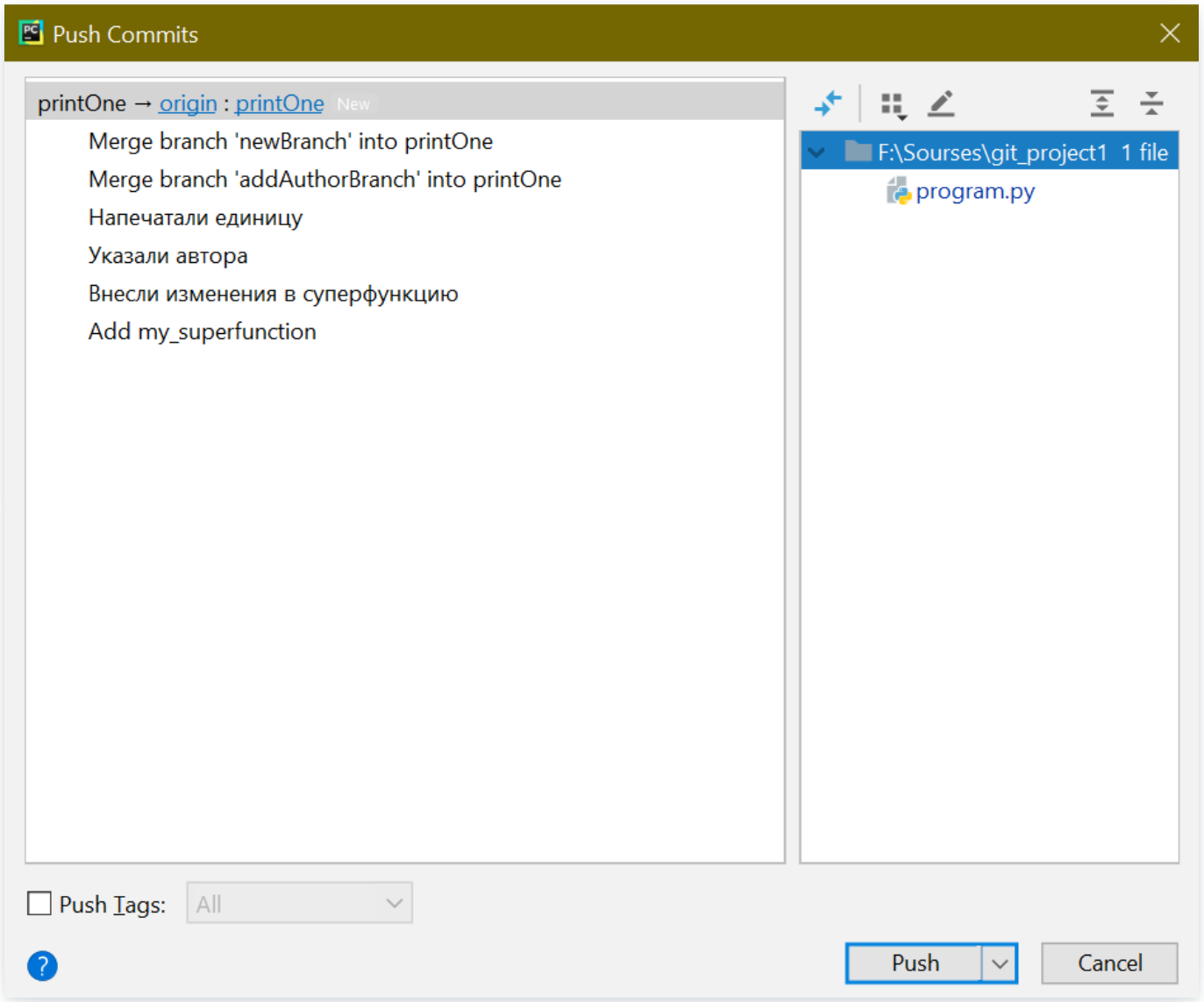
Для объединения нескольких веток (merge) в одну, нам необходимо перейти в ветку, в которую мы хотим влить изменения. Пусть это будет ветка printOne. Выберите в Log последний коммит ветки addAuthorBranch, кликните на ней правой кнопкой мышки и выберите команду Branch 'addAuthorBranch' → Merge into Current.



В Logs мы увидим, что у нас соединились две ветки в одну, а сам код — результат объединения этих веток.

Влейте ветку newBranch в printOne.

Если мы зайдем на GitHub, мы увидим, что изменения которые мы делали, там не отразились. Это нормально, ведь все изменения мы делали локально. Давайте запушим нашу ветку printOne:



После этого ветка printOne и коммиты из нее попадут в удаленный репозиторий. А у нашей ветки printOne на вкладке Logs появится фиолетовая бирочка и информация о том, что тот же код находится в удаленном репозитории в ветке origin/printOne.

3. Решение конфликтов

Не всегда объединение веток проходит гладко. Бывает, что программисты (или один программист в разных ветках) вносили изменения в одну и ту же строку кода, поэтому при объединении изменений возникнет конфликт, и объединение веток не будет считаться успешным до тех пор, пока этот конфликт не будет решен. Давайте смоделируем такую ситуацию.

Переключимся на ветку master и создадим там новый файл conflict_test.py вот с таким кодом:

```
print("Hello Git!")
print("Hello PyCharm!")
```

Добавим его к отслеживанию и закоммитим.

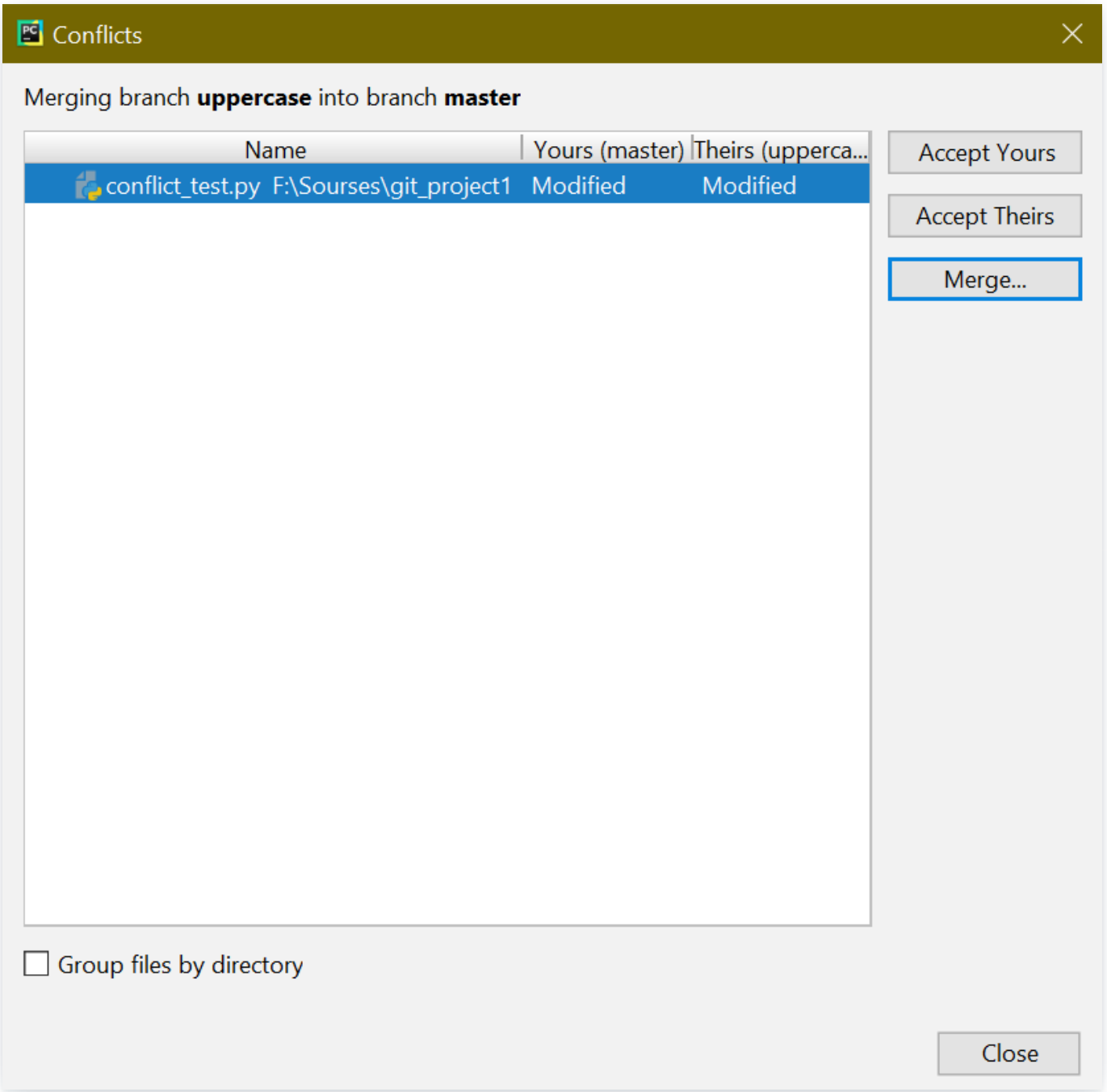
Создадим новую ветку lowercase (VCS → Git → Branches → New Branch, оставим галочку для переключения на новую ветку), и, стоя на ней, изменим наш файл следующим образом:

```
print("hello git!") # маленькие буквы
print("hello pycharm!") # маленькие буквы
```

Сделаем коммит. После этого переключимся на ветку master, убедимся, что файл вернулся к изначальной версии, и создадим еще одну ветку uppercase. Стоя на ней, изменим файл следующим образом:

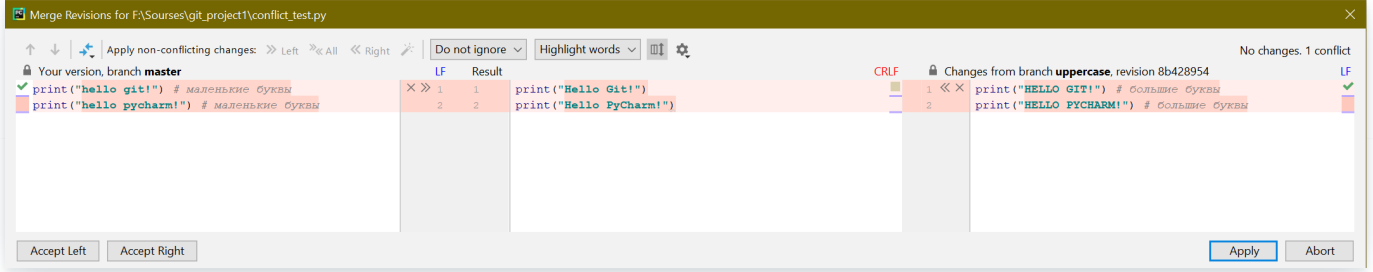
```
print("HELLO GIT!") # большие буквы
print("HELLO PYCHARM!") # большие буквы
```

Снова зафиксируем версию. Вернемся на ветку master и вольем в нее изменения из ветки lowercase. Слияние должно пройти успешно, а на ветке master появятся маленькие буквы. После этого выполним слияние с веткой uppercase. Теперь возник конфликт.



Нам показывается перечень файлов, в которых изменения затрагивают одни и те же строки. У нас есть несколько вариантов действий:

1. Accept Yours — оставить версию строк, которые находятся в текущей ветке (в которую вливаются изменения)
2. Accept Theirs — заменить все конфликтные строки на новую версию из вливаемой ветки
3. Merge — решить конфликт в ручном режиме. Это наиболее безопасный вариант



При выборе Merge откроется окно:

- Левая часть — текущая версия файла на ветке, в которую вливаются изменения
- Правая часть — версия файла на ветке, откуда вливаются изменения
- Центральная часть — версия, предлагаемая Git при попытке автоматического слияния, может быть основана на предыдущих версиях файла

Тут есть быстрые кнопки для принятия левой (Accept Left), правой (Accept Right) и центральной части (Apply). Также можно отменить слияние (Abort). Какую часть выбрать — решает разработчик. Еще он может редактировать центральную часть, создав новую версию файла. Решите конфликт любым способом.

4. Схема командной работы с репозиторием

Мы с вами уже научились работать с локальными и удаленными репозиториями, создавать, удалять и объединять ветки, фиксировать изменения в локальном и удаленном репозитории. Но во всех наших упражнениях ситуация была упрощенной: с репозиторием работал только один разработчик.

Когда разработчик один, то не особенно важно, как называть ветки, когда создавать новые ветки, в какое время и куда их вливать. Но все кардинально меняется, если с репозиторием работают несколько разработчиков:

- Многократно возрастает вероятность конфликтов в ветках
- Угадать, какие изменения сделаны в ветке, невозможно, если нет понятных имен веток и комментариев к коммитам
- Появляется необходимость как-то договориться о том, где мы храним полностью работающую программу, а где — ведем разработку и экспериментируем

И, наконец, нужно как-то работать с удаленным репозиторием, не мешая друг другу.

Для решения перечисленных проблем нужно, чтобы все разработчики проекта следовали единому своду правил работы с репозиторием.

Перечислим требования, которые предъявляются к такому своду:

1. Единый стандарт именования веток
2. Единый стандарт ветвления в репозитории
3. Единые правила изменения стабильной ветки (master)
4. Единые правила объединения репозиториев
5. Единые правила работы над проектными задачами

Такие своды правил называют flow.

В современной разработке чаще всего используются две популярные схемы (в пределах одного проекта рекомендуется выбрать и использовать только одну):

1. Классический подход Git Flow
2. GitHub Flow, адаптированный под коллективную работу на GitHub

Перед началом изучения методологий работы с репозиторием давайте зафиксируем терминологию — она идентична для всех основных flow:

- Стабильная ветка (Stable) — ветка, в которой программа в любой момент считается рабочей и хорошо протестированной. В идеале программу из этой ветки всегда можно передать заказчику — или, как часто говорят, *в продакшн* (production — рабочее состояние продукта). В большинстве методологий в качестве стабильной используется ветка master
- Ветка разработки (Dev) — ветка, относительно которой заводятся все новые изменения (именно от этой ветки отделяются новые ветки для разработки фич). Как правило, это ветка со всеми законченными в настоящий момент изменениями, без нестабильных изменений и изменений «в работе». Код в этой ветке чаще всего работоспособен, но не протестирован достаточно хорошо для передачи *в продакшн*. Из этой ветки часто собираются будущие релизы
- Релиз — процесс вливания ветки релиз-кандидата в мастер-ветку и выкладки ее в продакшн
- Релиз-кандидат/релизная ветка — ветка, где зафиксированы на момент релиза (выпуска новой версии продукта) все изменения, которые следующим шагом *поедут в продакшн*, то есть будут включены в новую версию. После сбора этой ветки вносить в нее изменения, кроме исправления ошибок (или *багов*), обычно запрещено
- Фича-бранч — ветка, в которой разработчик ведет разработку одной конкретной задачи или функциональности
- Хотфикс — ветка, отходящая от стабильной ветки или релиз-кандидата, нужная для исправления критического бага и обратного вливания в ту же ветку и все дочерние ветки

5. GitFlow

Самая старая и, вероятно, одна из наиболее распространенных методологий работы с репозиторием. Про нее часто спрашивают на различных собеседованиях при приеме на работу и применяют в относительно консервативных командах.

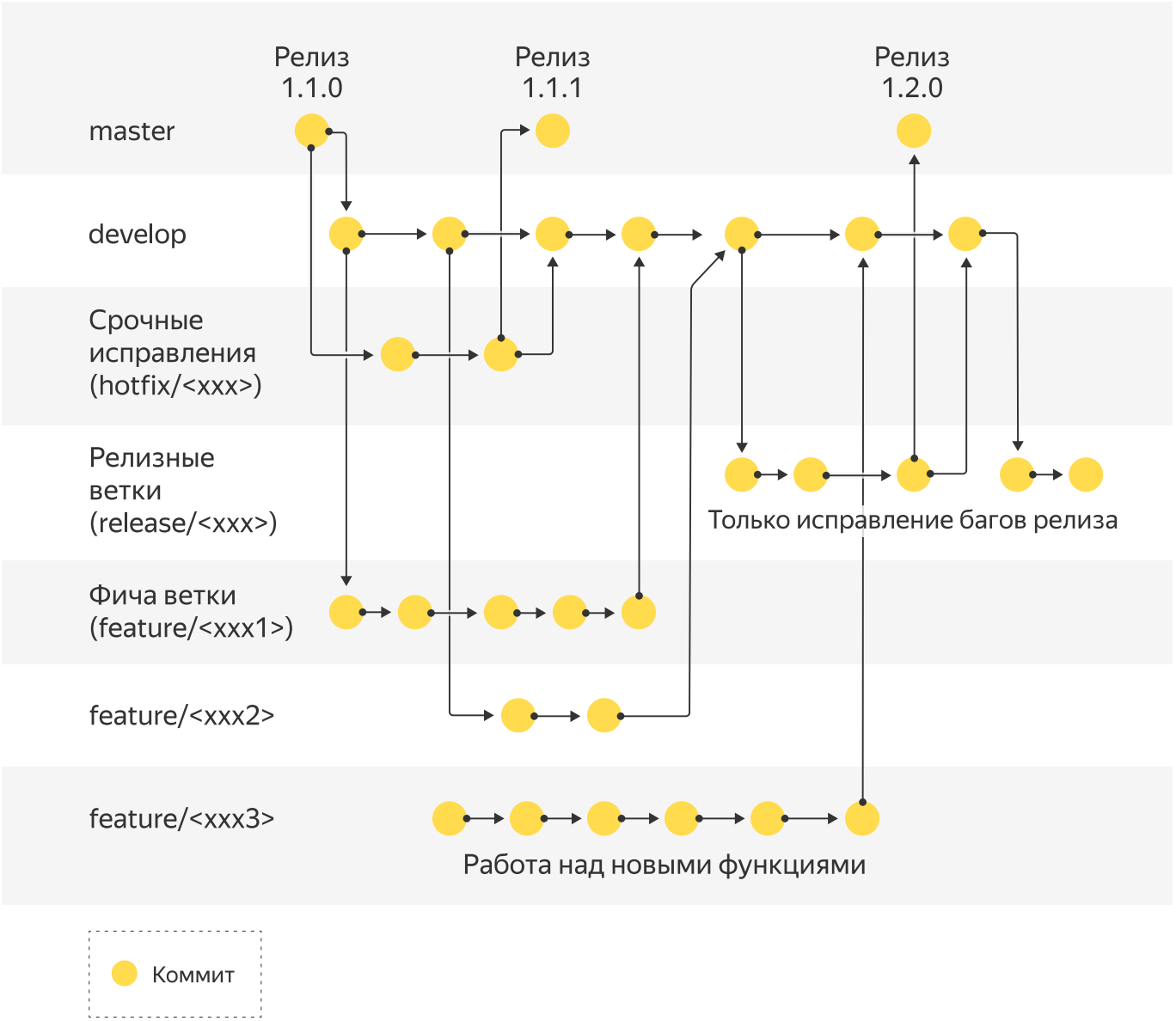
GitFlow использует описанные выше сущности следующим образом:

- Stable — ветка master
- Dev — ветка develop
- Фича-бранчи — ветки вида feature/<имя фичи/номер тикета в трекинг-системе> (это такие системы, в которых разработчикам ставят задачи и определяют сроки их выполнения, а разработчики отчитываются о результатах своей работы)
- Хотфиксы — ветки вида hotfix/<имя исправляемого бага/номер тикета в трекинг-системе>
- Релизные ветки — ветки вида release/<номер релиза/дата начала релиза>. Часто в качестве номера релиза используется symver-нотация: x.y.z, где x — версия программного продукта, версии с разной x-версией часто обратно несовместимы; y — номер релиза, если меняется только y — обратная совместимость не нарушается, чаще — исправляется старая функциональность или добавляется опциональная новая; z — номер патча, обнуляется при каждом обновлении y, часто используется для обозначения порядкового номера хотфикса. Подробнее о семантическом версионировании читайте **тут**
- Релиз/Тег — тег в ветке master, совпадающий по имени с именем релизной ветки (подробнее про теги читайте в инструкции git help tag)

В качестве примера трекинг-систем можно привести следующие:

1. **JIRA**
2. **Redmine**
3. **Яндекс.Трекер**

Общая схема работы с репозиторием в рамках GitFlow выглядит так:



Обратите внимание на то, как происходит выпуск **релизов**. Например, релиз с номером 1.1.1 получился только после внесения критичных правок, которые добавились и в **коммит** в ветке develop.

А версия 1.2.0 собралась уже из релизной ветки.

Предлагаем вам немного поработать по методологии GitFlow.

6. GitHubFlow

GitHubFlow — современная методология, которая распространена в компаниях, использующих систему GitHub. Она отлично совместима с процессом непрерывной поставки ПО и публичной работой над свободным программным обеспечением.

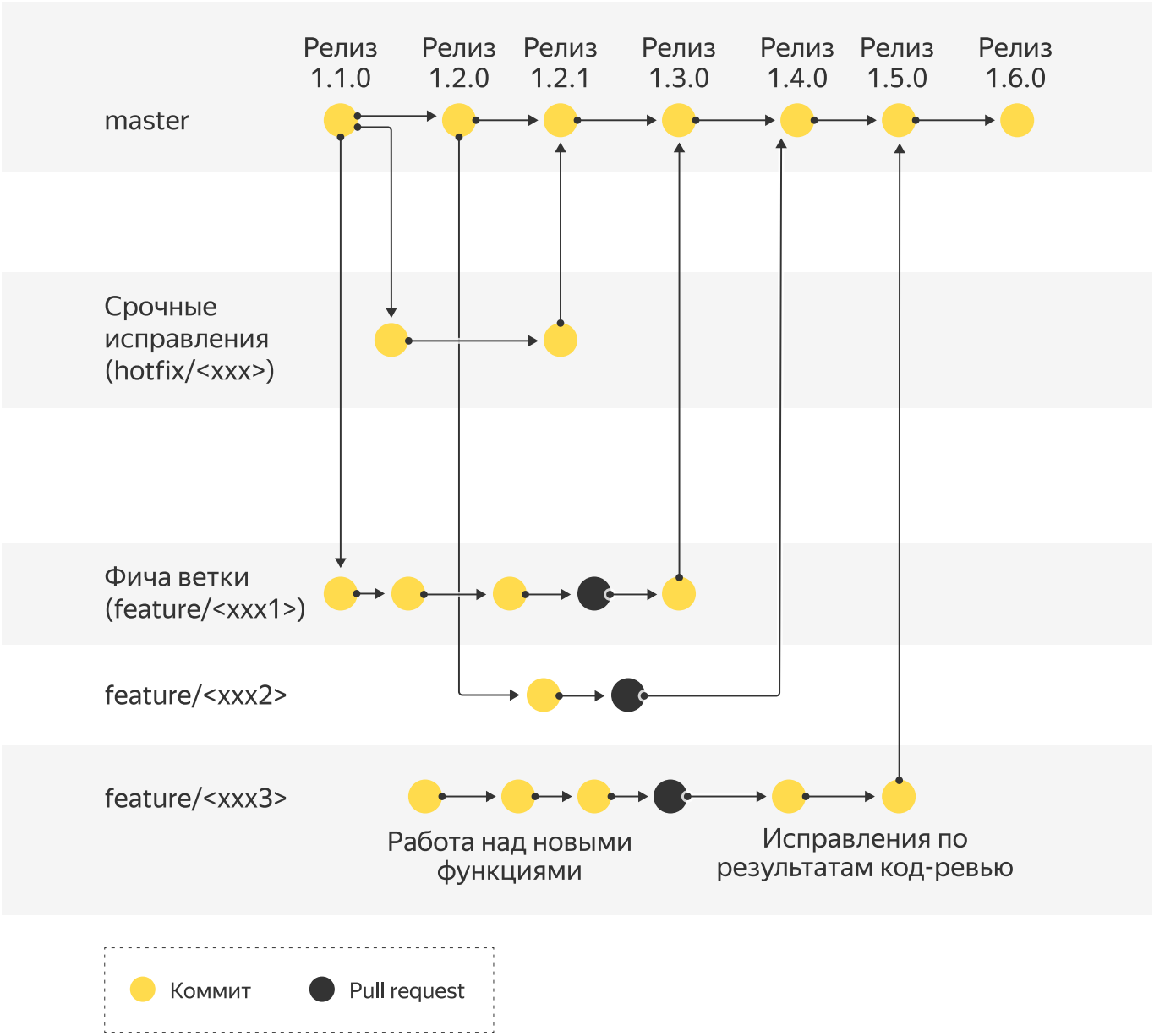
GitHubFlow оперирует следующими сущностями:

– Stable — ветка master

- Dev — сущность отдельно не выделяется. В качестве ветки, от которой осуществляется ветвление, используется ветка master
- Фича-бранчи — ветки вида feature/<имя фичи/номер тикета в трекинг-системе>
- Хотфиксы — ветки вида hotfix/<имя исправляемого бага/номер тикета в трекинг-системе>
- Релизные ветки — в виде отдельный сущности, как правило, не используются. GitHubFlow приветствует идеологию 1 фича = 1 релиз
- Релиз/Тег — тег в ветке master, совпадающий по имени с релизной веткой (подробнее про теги читайте в инструкции git help tag)
- Pull-request — сущность, специфичная для системы GitHub, заменяющая собой push в ветку master. После окончания работы над фича-веткой разработчик не делает merge в мастер-ветку, а создает **предложение** о внесении изменений в GitHub — так называемый pull-request. Далее основные разработчики проекта проверяют код пулл-реквеста (смотрят на корректность вносимых изменений или делают codereview) и либо принимают пулл-реквест (предварительно решив конфликты обычным слиянием), либо отклоняют пулл-реквест. Во втором случае изменения не попадают в ветку master

Подробнее почитать про GitHubFlow можно **тут**.

Общая схема работы с репозиторием в рамках GitHubFlow выглядит так:



7. Заключение

На этом мы закончим наш небольшой экскурс в системы контроля версий.

Вы научились работать в команде с удаленными репозиториями. Теперь у вас есть достаточно знаний для самостоятельного участия в разработке различных проектов.

На этом возможности PyCharm по работе с Git не заканчиваются. Сами разработчики JetBrains подготовили подробную **инструкцию** по работе с Git из PyCharm (на английском языке). Там вы найдёте много полезной информации и узнаете о дополнительных возможностях PyCharm, которые мы не успели рассмотреть за время урока.

Рекомендуем вам попробовать поработать с Git с помощью командной строки. Работая с репозиторием с помощью командной строки, программист строго контролирует все выполняемые операции, видит все происходящее в репозитории и может гибко настраивать функциональность Git под свои нужды: у каждой команды существует большое количество флагов, вы можете увидеть их с помощью git help <имя команды>. Кроме того, скорость работы командной строки значительно выше, чем у графических утилит. Графические утилиты, как правило, не реализуют в полной мере все функции системы Git. Тем не менее, с ними удобно и приятно работать, они позволяют выполнять все часто используемые команды. Большое преимущество работы с Git через IDE: не нужно переключаться между окнами и приложениями, чтобы сделать коммит или получить изменения с сервера.

Кроме того, мы не затронули целый ряд полезных тем, касающихся работы с репозиториями. Поэтому предлагаем вам изучить их самостоятельно.

1. **Форки** в GitHub — очень мощный инструмент для работы с публичными репозиториями, популярный в opensource community
2. **Rebase** — еще один очень популярный способ объединения изменений, удобный при работе над достаточно большими фичами
3. **Cherry-pick и reset** — инструменты для работы с историей коммитов: извлечением какого-либо коммита, или откатом последних коммитов
4. Так же мы рекомендуем пошаговый **самоучитель** по Git для закрепления пройденного материала