Средства автоматизации процессов разработки ПО

Введение

Цель

Получить навыки коллективной разработки с использованием таких средств автоматизации, как системы контроля версий, системы управления требованиями и изменениями

Обозначения

`так обозначается команда, если она встречается в тексте`

```
так, если она выделена в отдельный блок
это команда # а после решетки -- комментарий
```

Чтобы отличить команду от ее вывода, если они встретсятся вместе, то команда будет выделена `>`, а результат комментарием(#):

```
> команда
# результат её выполнения
```

Системы контроля версий

Системы контроля версий (СКВ, VCS) ведут учет изменений файлов, а так же предоставляют возможность перемещаться между различными состояниями проекта. Это означает, что при потере или повреждении файлов или просто неудачной попытке решить поставленную задачу, вернутся к исходному состоянию не составляет труда при использовании СКВ.

Типы СКВ

Локальные (ЛСКВ)

База данных изменений хранится на компьютере разработчика.

+ нет взаимодействия по сети, сопутствующих задержек
+ не нужен доступ к другим компьютерам, чтобы внести изменения
- при потере базы данных из-за повреждения файла базы данных СКВ все данные учета версий теряются
- сложная координация действий между членами команды

Централизованные (ЦСКВ)

База данных изменений хранится(назовем ее *“главный репозиторий”*) на единственном компьютере, к которому имеют удаленный доступ разработчики.

+ при потере данных одним или даже несколькими разработчиками, у них имеется возможность восстановить проект с помощью главного репозитория
+ простая координация действий между членами команды
- при повреждении данных главного репозитория все данные учета изменений для всех членов команды теряются

Распределенные

Чаще всего остается разделение на серверы и клиенты. Под сервером понимается устройство или сервис, хранящий изменения и не используемый для разработки. В отличие от ЦСКВ, клиенты, получая от сервера изменения, получают впридачу и копию базы данных. Таким образом, на всех компьютерах разработчиков есть копия базы данных, пригодная, чтобы восстановить состояние сервера или другого клиента.

+ наибольшая устойчивость и сохранность данных учета
+ возможность вносить изменения только в личную копию БД с последующей синхронизацией с сервером (или серверами)
+ достаточно простая координация между разработчиками (немного сложнее, чем в ЦСКВ)
- необходимость хранить локальную копию репозитория

Система контроля версий Git

Для обучения использованию СКВ будем использовать Git. Это распределенная система. Несмотря на это, она достаточно эффективно использует дисковое пространство, нивелируя недостатки данного типа СКВ. Git в состоянии поддерживать в рамках одного проекта параллельную разработку нескольких тысяч направлений, которые изолированы друг от друга и при должной координации способны не мешать друг другу.
Одним из главных отличий Git от других СКВ является система *веток (branch)*. Именно она предоставляет возможность параллельной работы над проектом.
Есть два способа сделать код проекта учтенным в Git: создать проект, а затем инициализировать в нем репозиторий и создать *коммит*[1] или клонировать (от названия команды `git clone`) у другого члена команды или с сервера.
Все эти и многие другие действия совершаются с помощью программы git. Эта программа является интерфейсом между пользователем и СКВ. Существует множество графических оболочек для нее, но все они предоставляют некоторое подмножество встроенных команд Git. В ходе работы над проектом нужно использовать CLI (консольную) версию.
Сама по себе команда `git` лишь выводит собственную справку, она служит обращением к системе. Чтобы совершить какие-либо действия, нужно обратиться к git с командой, например:

```
git status
```

чтобы вывести текущее состояние проекта и СКВ. git имеет достаточно много команд, однако, каждая из них снабжена подробным встроенным руководством:

```
git help <command> # например:
git help commit
```

Установка

В качестве сервера будет использоваться GitHub, речь о котором пойдет позже. Каждому члену команды необходимо установить гит. Способы установки для разных платформ, а так же подробные руководства можно найти на git-scm[2] (Введение → Установка).

Настройка

После установки необходимо провести настройку. В книге по ссылке выше, можно так же найти и подробное руководство по настройке. Надо только сказать, что ваше имя пользователя при вводе `git config --global user.name` должно быть настоящим.
Так же настоятельно рекомендуется сменить текстовый редактор по умолчанию на что-то вроде Sublime Text или хотя бы Notepad (блокнот), чтобы избежать неразберихи с редактором vi, который git использует по умолчанию.

```
git config --global core.editor subl # для Sublime Text
```

---

[1] В Git учтенное состояние папки проекта называют “commit”.
[2] Это электронная версия второго издания книги “Pro Git” [1]

Основные принципы и понятия

Репозитории

Проект, в котором проинициализирован Git называют в рамках СКВ репозиторием. Это относится и к копии проекта на сервере, и к копии у разработчика.
Чтобы создать репозиторий у себя из рабочей папки проекта выполняют:

```
git init
```

Чтобы скопировать удаленный репозиторий к себе:

```
git clone <url>
```

После выполнения какой-то из этих команд у вас будет локальный репозиторий Git.

Коммит

Англ. commit. Зафиксированное состояние проекта. История проекта в Git представляет собой последовательность коммитов, каждый из который имеет ссылку на родителя (иногда более, чем одного, об этом позже).

Состояния

В Git существует три состояния файла:

1) не учтен, не индексирован (untracked);
2) подготовлен к коммиту, далее просто подготовлен (staged);
3) зафиксирован (committed).

Состояние “не учтен” означает, что для Git это новый файл, то есть он не был зафиксирован ни в одном состоянии проекта или был удален из индекса(`git rm --cached`).
Подготавливают файлы (а если быть точнее, изменения в файлах, строки), чтобы затем их зафиксировать. Файлы не фиксируются сразу, потому что каждый commit должен быть осмысленным и решающим какую-то задачу, для чего может понадобиться модификация сразу нескольких файлов. Подготовка делается командой

```
git add <filename>
```

```
и л и
```

```
git add -A # с р а з у  в с е  н е у ч т е н н ы е  и з м е н е н и я.
```

Далее файлы фиксируют, то есть создают снимок текущего состояния проекта. При этом фиксируются только добавленные в предыдущем шаге изменения. Можно проигнорировать файлы или отдельные строки, которые логически не связаны с решением поставленной задачи. Скажем, вы работаете над исправлением ошибки в какой-то функции, но вместе с тем исправили или добавили еще что-то.
Фиксация подготовленных файлов делается командой

```
git commit
```

Принято снабжать коммиты сообщениями, чтобы дать понять другим и себе, что делает данное изменение. Выполнение команды выше вызовет ваш текстовый редактор и попросит вас ввести это сообщение. Можно написать сообщение сразу в команде:

```
git commit -m “message here”
```

Удаленный репозиторий

Обычно для координации используют один компьютер в качестве сервера, на который все записывают изменения и скачивают оттуда последние версии.
Доступ осуществляется с помощью URL, но в git предусмотрена система сокращенных названий:

```
git remote  # перечислить известные URL и именами и ветками
git remote add <имя> <URL>  # добавить именованную URL
```

Чаще всего используется название `origin`, в котором, впрочем, ничего особенного нет, просто при выполнении команды `git clone <url>` к разработчику попадает репозиторий, для которого уже установлено сокращение origin → URL.

Ветки

Считается основным отличием Git от прочих СКВ. Ветка — это обособленная серия коммитов. Под обособленностью понимается следующее: с момента создания ветки до момента ее слияния с какой-либо другой веткой, изменения в ней никак не влияют на какую либо другую ветку. Например, вам нужно устанить баг за номером N. Вы создаете ветку “issue-N” и пытаетесь решить свою задачу. При этом на каком-то этапе ошибка устранена, но тесты показывают, что в другом месте проекта, на которое вы повлияли, теперь что-то идет не так. То-есть проект в целом оказался в нерабочем состоянии. Но вся деятельность ведется в отдельной ветке и на сервере в то же время доступен работоспособный проект. В конце концов вы добиваетесь прохождения всех тестов и производите слияние своей ветки с родительской, после чего можете удалить свою ветку.

GitHub

https://github.com
GH — это сервис, предоставляющий Git-репозитории с настройками доступа, баг-трекер(см. далее) и некоторое подобие социальной сети. Все это можно использовать для координации действий своей команды для выполнения проекта.
Для использования требуется бесплатная регистрация. Со всеми возможностями, а так же актуальными инструкциями можно ознакомиться прямо на сайте. Так-же инструкции и подробности на русском языке можно найти в переведенной книге Pro Git [1].

Виртуальное рабочее место

aka Виртуальная Среда Разработки

Часто бывает так, что на компьютере разработчика проект собирается и работает без ошибок, а на целевой машине (где предполагается использовать данный код) —нет. Чтобы не попадать в такие ситуации, стараются тестировать продукт на целевой машине, а если такой возможности нет, то на имитаторах.
К таким имитаторам можно отнести и виртуальные машины. Во многих случаях они являются удовлетворительной имитацией целевой машины. А в отдельных случаях можно вести разработку прямо внутри виртуальной машины, не только тестировать на ней.
В последнее время так же стало популярным использование виртуальных контейнеров. Это можно описать как весьма легковесную виртуальую машину. Одной из таких систем является **Docker**. Подробности можно найти на сайте http://docker.com. Вкраце, это программа, управляющая виртуальными контейнерами. С помощью docker можно:

- имитировать удаленные сервисы, такие как базы данных
- вести разработку внутри контейнера, который имитирует целевую машину
- можно хранить проекты в контейнерах, подобно образу жесткого диска, а затем разворачивать на нескольких машинах

Docker имеет встроенный учет изменений в контейнерах, хранит состояние контейнеров, позволяет обмениваться файлами и сетевыми пакетами с внешней ОС.
Использование вместе с СКВ дает возможность фиксировать состояния не только проекта, но и внешней среды его исполнения.

Автоматизированное тестирование

Тестирование, которое проводится при минимальном участии человека. Идеальное автоматизированное тестирование сводится к автоматическому: все требования к проекту покрываются тестами, которые требуют только запуска.
Однако, чем выше уровень тестирования, тем больше временных и умственных затрат требуется на написание автоматизированного теста. Так, написать юнит-тест куда быстрее и проще, чем тест, проверяющий интерфейс удаленной подсистемы.
В итоге получается, что чем выше уровень тестирования, тем выше человеческая вовлеченность.
Для организации автоматизированных тестов используют встроенные в среды разработки или обособленные фреймворки, так же просто скрипты или даже другие программы. Необходимо, конечно, чтобы система, которая проводит тестирование, сама была должным образом отлажена.
Далее предполагается, что используется подход TDD или разработка через тестирование, но это не обязательно.
Такими системами являются, например, DUnit для C++ и PHPUnit для PHP.
Метод для юнит-тестирования следующий. Исходя из списка требований или утвержденной архитектуры проекта пишут файл юнит-теста, который проверяет нужный модуль через непосредственное обращение к нему. Например, есть модуль Math с методом sqrt(), который извлекает квадратный корень из своего аргумента. Псевдокод проверки такой:

```
подготовка(
        m = Math
)
тест_Math_sqrt(
        убедиться_что_равно(Math.sqrt(16), 4)
        выбрасывает_исключение(Math.sqrt(-1))
        выбрасывает_исключение(Math.sqrt(“a string”))
)
очистка(
        // в данном случае нет. Обычно тут чистят базу данных и т.п.
)
```

Баг-трекеры

Системы учета ошибок и пожеланий.
Нужны для координации разработчиков между собой и тестировщиков. Могут использоваться для управления проектом как средство постановки задач.
В этой системе работают с такими сообщениями как bug(ошибка) и issue(проблема, но также к этому типу относят пожелания или улучшения). Их состав обычно влючает в себя:

- уникальный номер и дату
- короткое описание дефекта или пожелания
- кто оставил сообщение (сообщил об ошибке)
- версия проекта, в которой обнаружен дефект
- ситуация, условия, в которых обнаружена ошибка
- статус (открыт, закрыт, отложен, назначен, тестирование)

Последовательность действий для случая, когда баг-трекер используют и для управления проектом в том числе, следующая:

1) выявляется набор задач;
2) задачи распределяются по членам команды, каждой задаче выставляют статус “назначена”. При этом система отображает когда и кому назначена задача;
3) после того, как разработчики выполняют задание, начинается тестирование изменений. При этом, если тестированием занимаются другие люди, имеет смысл сменить статус на “тестирование” и обновить назначение члена команды;
4) после успешного прохождения тестов, задаче выставляют статус “закрыта”. При необходимости исправлений в результате неуспешного тестирования возвращаются к шагу два.

Все написанное справедливо и для устранения ошибок.

Системы непрерывной интеграции

Непрерывная интеграция(НИ) — практика разработки ПО, при которой совершаются частые сборки и тестирование проекта. Целью является уменьшение задержек на этапе интеграции проекта, в котором часто возникают задержки из-за различных несовместимостей модулей проекта и прочих ошибок. Далее система непрерывной интеграции СНИ.

Обычно работа СНИ заключается в автоматизации сборки и проведении автоматических тестов с последующим оповещением заинтересованных лиц о результатах.

Многие СНИ обеспечивают интеграцию с СКВ: при публикации очередной (предположительно стабильной или работоспособной) версии проекта на сервер СКВ, СНИ автоматически получает образец проекта и проводит необходимые действия.

Практическая часть

Git shell

Для выполнения задания работать нужно будет в Git Shell. Для Windows это отдельный исполняемый файл, который запустит командную строку, в которую интегрированы некоторые возможности posix-совместимых командных оболочек (стандартные команды, перенаправление потоков). Для Linux и Mac OS git — просто еще одна команда терминала.

Для работы понадобится всего несколько не-git команд:

```
# где я:
pwd
# перейти в папку:
cd <имя>
cd .. # на уровень выше
# список файлов текущей директории:
ls -al
# создать файл, если еще нет
touch <имя>
```

Про именование файлов: если имя начинается с точки, то файл (или папка) *скрытые*. Например файл `.gitignore` и папка `.git`.

Задание

Каждый участник группы как можно раньше в процессе разработки должен:

1) установить на компьютер программу git;
2) провести базовую настройку;
3) создать аккаунт на GitHub и привязать к нему свои данные git, указать реальные имя и фамилию;
4) из состава программистов команды выбрать одного, кто будет заниматься координацией в системе контроля версий, назовем его руководителем интеграции(РИ);
5) вступить на GitHub в организацию ‘mirea-fcyb-2015’; для этого отправить ссылку на свой профиль (что-то вроде http://github.com/vasya), ФИО и добавить, что вы РИ (если это так) на почту создателю организации (консультанту).Через некоторое время вас пригласят в организацию (оповестят письмом);
6) после того, как руководитель команды присоединится, он должен внести требования и расставить задачи;
7) программистам зафиксировать текущий код проекта в git.

В зависимости от того, в какой стадии находится разработка до начала настройки СКВ, возможны случаи, оговоренные ниже.

Проект находится на стадии проектирования и файлов исходного кода еще нет ни у кого из членов команды

1. В группе уже созданы пустые репозитории для всех команд. РИ необходимо клонировать к себе нужный репозиторий:

```
# это выполняется в папке, в которой планировалось разместить папку с проектом
git clone https://github.com/mirea-fcyb-2015/<название репозитория>
```

2. В папке, где выполнена команда появится папка с названием репозитория, перейти туда:

```
cd <название>
```

3. Удостовериться, что все прошло нормально:

```
> git status
# On branch master
#
# Initial commit
#
# nothing to commit (create/copy files and use "git add" to track)
> git remote -v
# origin  https://github.com/mirea-fcyb-2015/project-1 (fetch)
# origin  https://github.com/mirea-fcyb-2015/project-1 (push)
```

4. Создать файл “`.gitignore`” и наполнить в зависимости от того, какой именно язык и среда разработки используется при разработке. Этот файл позволяет исключить из контроля версий собранные исполняемые файлы, конфигурацию среды разработки конкретного разработчика, логи, временные файлы и подобное, что не требуется учитывать и передавать друг другу.

```
touch .gitignore # это не расширение, файл так и называется
```

Файл создан. Сейчас он пустой и его нужно наполнить шаблонами имен файлов, которые не будут далее учтены. Для этого открыть его текстовым редактором по выбору и наполнить содержанием из приложения.

.gitignore для RAD Studio

.gitignore для Visual Studio

.gitignore для PHP зависит от выбора фреймворка, среды разработки. Если используется Visual Studio, то можно взять соответствующий .gitignore.

Если по каким то причинам такие исключения не подходят, то можно редактировать эти, или поискать более подходящие на https://github.com/github/gitignore.

5. Создать файл README.md (это файл разметки Markdown. GitHub широко использует его для форматирования подобных документов, а так же комментариев пользователей) и заполнить его (открыв в редакторе). Этот файл должен содержать самую базовую и необходимую информацию для человека, который не представляет, что это за проект.

Например(решетка в md не комментарий, а заголовок: h1 → #, h3 → ###):

Project-1

Программа, которая делает то-то и то-то

build

Проект для среды Embarcadero RAD Studio <версия такая-то>

6. Убедиться, что файлы действительно изменены и еще не добавлены в Git:

```
> git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       README.md
#
# nothing added to commit but untracked files present (use "git add" to #track)
```

7. Подготовить файлы для Git:

```
git add README.md
git add .gitignore
# или чтобы добавить все, что еще не добавлено:
git add -A
```

8. Убедиться в том, что файлы подготовлены к коммиту:

```
> git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   .gitignore
#       new file:   README.md
```

9. Создать коммит, тем самым зафиксировать изменения:

```
> git commit -m ‘первый коммит’
# [master (root-commit) b34ea06] первый коммит
# 2 files changed, 6 insertions(+)
# create mode 100644 .gitignore
# create mode 100644 README.md
```

Вся дальнейшая работа над проектом ведется в этой папке.

10. Создать проект в нужной среде разработки в данной папке и сохранить его.
11. Подготовить и зафиксировать изменения:
    `git add -A`
    `git commit -m ‘создал проект’`
12. Опубликовать изменения на GitHub:
    `git push origin master`
13. Перейти на страницу проекта и удостовериться, что код и два созданных ранее файла появились в зафиксированных изменениях на GitHub, то есть теперь доступны всем. Страница проекта находится по адресу https://github.com/mirea-fcyb-2015/<название репозитория>
14. **Всем остальным программистам** необходимо получить себе такую рабочую папку и вести работу только в ней. Перейдя в папку, где предполагается иметь папку с проектом выполнить:
    `git clone`
    https://github.com/mirea-fcyb-2015/<название репозитория>
    Это автоматически создаст папку проекта, скопирует все то, что сделал РИ и установит удаленный репозиторий для git, под названием **origin**.

Исходный код проекта уже есть у кого-то из программистов

В данном случае сначала РИ (если код у РИ, тот временно перемещает его в какую-нибудь другую папку, не ту, где предполагается разрабатывать) повторить **шаги 1 .. 9** для предыдущего случая, а потом, если код не у РИ, тот должен получить его от другого члена команды и положить в папку с проектом, где к этому времени уже лежать README и .gitignore. Проект должен лежать не в подпапке, а прямо там. После этого РИ выполняет **шаги 11, 12 и 13**.
Остальные программисты, в т.ч. у кого изначально был код (он получит копию своей работы от РИ), выполняют **шаг 14**.

Работа уже начата и разные версии исходных кодов есть у нескольких программистов

1. РИ поступает как в предыдущем случае, код можно взять у любого из программистов.
2. **Остальные программисты** временно перемещают имеющийся у них код в какие-либо другие папки (если название совпадает с названием проекта) и выполняют **шаг 14**.
3. Все кроме РИ, чьи версии исходного кода отличаются, копируют свои файлы (обратно) в папку проекта и выполняют:
   `git status`
   Вывод команды укажет на “Untracked files” (неучтенные) или “Modified” (файл уже был отслежен ранее, но изменен и не подготовлен к коммиту). “Nothing to commit, working directory clean” означает, что версии совпадают.

Далее начинается рабочий процесс внесения изменений.

Рабочий процесс внесения изменений

На данный момент в проекте есть единственная ветка *master.* Пусть далее эта ветка будет содержать только код, который точно работает. Пишутся тесты, которые подтверждают это.

Сейчас нужно остановиться и уяснить кое-что про ветки. Ветки в git — это, как уже говорилось, обособленная серия коммитов. Правда Git обращается с ветками как с указателями на последний коммит. Вообще, в Git есть 3 вида подобных указателей:

- ветка, например *master,* или *feature, или vasya;*
- HEAD — указатель на тот коммит, в состоянии которого находится *сейчас* папка проекта;
- удаленная(remote) ветка, указатель на состояние ветки в удаленном репозитории, например: origin/master.

Так как каждый коммит в git ссылается на предыдущий (иногда больше, чем один, если он создан в результате `git merge`), то восстановить любой коммит в ветке не составляет труда.
Важно следующее: до того момента, пока не выполнена команда `**git push**` все изменения в ветках остаются локальными для разработчика, больше их никто не видит.

В рамках этой работы, можно (и нужно) создавать несколько удаленных веток, однако необходимо, чтобы веткой *master* занимался только РИ, и чтобы там был только рабочий и протестированный код.

Итак, программист сделал собственную ветку прямо от первого коммита (где был добавлен проект). Назвал ее, скажем, **dev-1**. Потом сделал в ней некоторые изменения протестировал их, *зафиксировал*, и хочет теперь зачесть. Для этого они должны оказаться в ветке (удаленной, то есть в интернете, на GitHub) *origin/master.*
Последовательность действий программиста такова:

1. Обновить локальную версию репозитория до актуальной, чтобы учесть все прошедшие изменения:`git fetch` `git merge origin/master` Возможно, это вызовет конфликт слияния, если другие разработчики поменяли те-же самые места, что и рассматриваемый. На этот случай придется разрешить конфликты вручную:
   1.1. `git status`
   1.2. Найти файлы, напротив которых “both modified”, каждый из них открыть текстовым редактором и найти строчки вида

```
<<<<<<< HEAD
my_function(void *args)
=======
my_function(int a, char *b, void *other)
>>>>>>> origin/master
```

   Сверху изменения программиста, снизу — изменения других. Необходимо выбрать и оставить один из вариантов, удалив все посторонние символы (<<< >>> ===).
   1.3. `git status`
   1.4. Убедиться, что все подобные конфликты разрешены (нет больше “both modified”)
   1.5. `git add -A`
   1.6. `git commit`
   1.7. убедиться, что тесты проходят нормально, ничего не сломалось. Поправить, если необходимо.
   Текст сообщения изменить на “провел слияние master и (что я сделал)”
2. Фиксированные изменения залить на GitHub при помощи **push**:

```
git push origin dev-1
```

3. Информировать РИ (письмом, по телефону, как угодно) о том, что его ветка (лучше напомнить название) содержит изменения и готова к слиянию.

РИ, получив уведомление, выполняет следующее:

1. Получает с GitHub информацию об изменениях:

```
git fetch
```

2. Уточняет, что за изменения, в какой точно ветке с помощью:
   `git log --all --graph --decorate --remotes`
   (находит на рисунке ответвление, которое представляет нужную ветку, напротив последнего коммита будет надпись вроде
   “* 931d3e6 2015-09-28 | created file (origin/dev-1, dev-1) [vasyan]”.
   Если это так, значит разработчик выполнил контроль версий со своей стороны верно и можно продолжить.
3. Уточняет, что находится в (локальной) ветке *master*:
   `git branch`
   Если звезда стоит не напротив *master,* а какой-либо другой ветки, выполняет: `git checkout master`
4. Выполняет слияние веток (это создаст коммит):
   `git merge origin/dev-1`
5. Так как слияние локально, отправляет этот коммит в удаленный *master*:
   `git push origin master`

Теперь коммит со слиянием находится на сервере и изменения разработчика учтены.

Команды git управления ветками

```
# сменить рабочую ветку
git checkout <название ветки>

# создать ветку и переключиться на нее
git checkout -b <название ветки>

# получить последнюю версию изменений, но остаться в текущей ветке
git fetch

# соединить ветки (внести изменения из одной ветки в другую)
# находясь в ветке1 с намерением получить изменения ветки модуль1
git merge модуль1

# скачать информацию о последних объявлениях других веток
git fetch

# посмотреть состояние (коммиты включая авторов, сообщения и ветки)
git log --all --graph --decorate

# только сообщения коммитов (проще увидеть структуру)
git log --all --graph --oneline --decorate
```

Критерии оценки выполнения

Слово “собирается” в случае интерпретируемых языков понимать как “не выдает ошибок выполнения”.

- все коммиты содержат осмысленные сообщения
- каждый новый коммит в *master* является слиянием с веткой одного из программистов, и добавляет новую функциональность (исправляет существующую)
- код покрыт тестами: юнит-тесты новых модулей, интеграционные тесты. К новому коммиту в ветке *master* должны быть тесты, доказывающие что модуль, функция или вообще код, который он добавляет

- рабочий
- не выводит из строя остальные компоненты проекта

То есть ситуации, в которой в новом коммите перестает работать функционал, который в более раннем коммите работал быть не должно.

Материалы

Pro Git — Scott Chacon, Ben Straub. Published by Apress.

Доступна для чтения и скачивания по адресу: https://git-scm.com/book/ru/v2

.gitignore для RAD Studio

```
#########################
## Embarcadero RAD Studio
#########################

## Ignore RAD Studio temporary files, build results, and

# User-specific files

__history/
*.~*
*.bak
*.bpl
*.cfg
*.dcu
*.dll
*.exe
*.identcache
*.local
*.stat
*.xml
*.zip
```

.gitignore для Visual Studio

```
#################
## Visual Studio
#################

## Ignore Visual Studio temporary files, build results, and
## files generated by popular Visual Studio add-ons.

# User-specific files
*.suo
*.user
*.sln.docstates

# Build results
```

```
[Dd]ebug/
[Rr]elease/
x64/
build/
[Bb]in/
[Oo]bj/

# MSTest test Results
[Tt]est[Rr]esult*/
[Bb]uild[Ll]og.*

*_i.c
*_p.c
*.ilk
*.meta
*.obj
*.pch
*.pdb
*.pgc
*.pgd
*.rsp
*.sbr
*.tlb
*.tli
*.tlh
*.tmp
*.tmp_proj
*.log
*.vspscc
*.vssscc
.builds
*.pidb
*.log
*.scc

# Visual C++ cache files
ipch/
*.aps
*.ncb
*.opensdf
*.sdf
*.cachefile
```

```
# Visual Studio profiler
*.psess
*.vsp
*.vspx

# Guidance Automation Toolkit
*.gpState

# ReSharper is a .NET coding add-in
_ReSharper*/
*.[Rr]e[Ss]harper

# TeamCity is a build add-in
_TeamCity*

# DotCover is a Code Coverage Tool
*.dotCover

# NCrunch
*.ncrunch*
.*crunch*.local.xml

# Installshield output folder
[Ee]xpress/

# DocProject is a documentation generator add-in
DocProject/buildhelp/
DocProject/Help/*.HxT
DocProject/Help/*.HxC
DocProject/Help/*.hhc
DocProject/Help/*.hhk
DocProject/Help/*.hhp
DocProject/Help/Html2
DocProject/Help/html

# Click-Once directory
publish/

# Publish Web Output
*.Publish.xml
*.pubxml

# NuGet Packages Directory
```

```
## TODO: If you have NuGet Package Restore enabled, uncomment the next line
#packages/

# Windows Azure Build Output
csx
*.build.csdef

# Windows Store app package directory
AppPackages/

# Others
sql/
*.Cache
ClientBin/
[Ss]tyle[Cc]op.*
~$*
*~
*.dbmdl
*.[Pp]ublish.xml
*.pfx
*.publishsettings

# RIA/Silverlight projects
Generated_Code/

# Backup & report files from converting an old project file to a newer
# Visual Studio version. Backup files are not needed, because we have git
;-)
_UpgradeReport_Files/
Backup*/
UpgradeLog*.XML
UpgradeLog*.htm

# SQL Server files
App_Data/*.mdf
App_Data/*.ldf
```