

Инструменты разработки ПО



Кирилл Корняков (Itseez, ННГУ)
17 Августа 2015

Содержание

1. Введение
2. Кросс-платформенная разработка: CMake
3. Коллективная работа с кодом: Git
4. Автоматическое тестирование: Google Test, Travis-CI

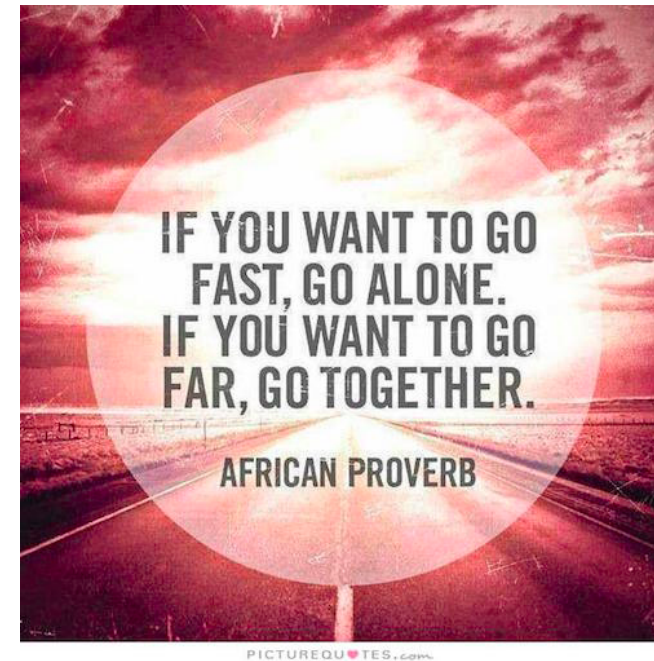
Ожидания начинающего программиста

- Мне будут давать сложные задачи, которые способен решить только я
- Я буду записываться один, быстро писать кучу гениального кода
- Когда я закончу код, моя задача будет выполнена
- Код сразу заберут и последуют одни восторженные отзывы
- Мне сразу дадут еще более сложную и интересную задачу



А вот что ждут от профессионала

- Работающий код
 - *Не только на его компьютере*
 - *Компилирующийся под Embedded Linux, iOS и Android*
 - *Корректно обрабатывающий всякие нелепые ситуации*
- Качественный код
 - *Чистый, короткий, понятный, с хорошими именами*
 - *Набор автоматических тестов*
 - *Приемлемо документированный*
- Присланный по правилам
 - *Исключительно через СКВ*
 - *Маленькими осмысленными порциями (они видят ли его не понимают!)*



Черты современного процесса разработки ПО

- Сложность проектов
 - *Работа ведется в командах*
 - *Коллективы часто распределены*
 - *Большой объем унаследованного кода (legacy)*
- Динамичная конкурентная среда
 - *Требования к ПО быстро меняются*
 - *Высокие ожидания к стабильности ПО*
 - *Готовность выпустить релиз в любой момент*
- Разнообразие конечных платформ
 - *Различные операционные системы*
 - *Различные виды приложений*

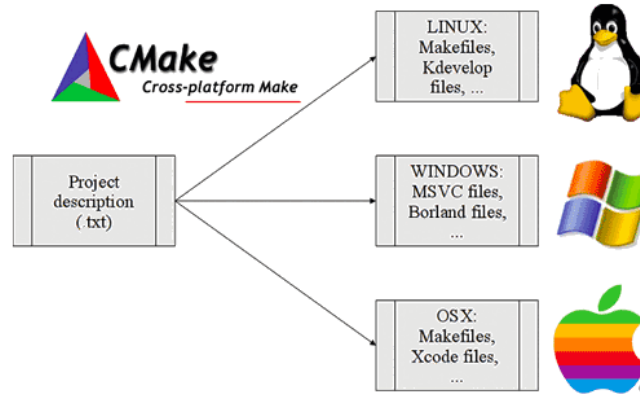
Про что мы сегодня поговорим

- Кросс-платформенная разработка
- Коллективная разработка
 - *Использование СКВ*
 - *Peer code review*
- Тестирование
 - *Автоматические тесты*
 - *Непрерывная интеграция*



CMake
Cross-platform Make

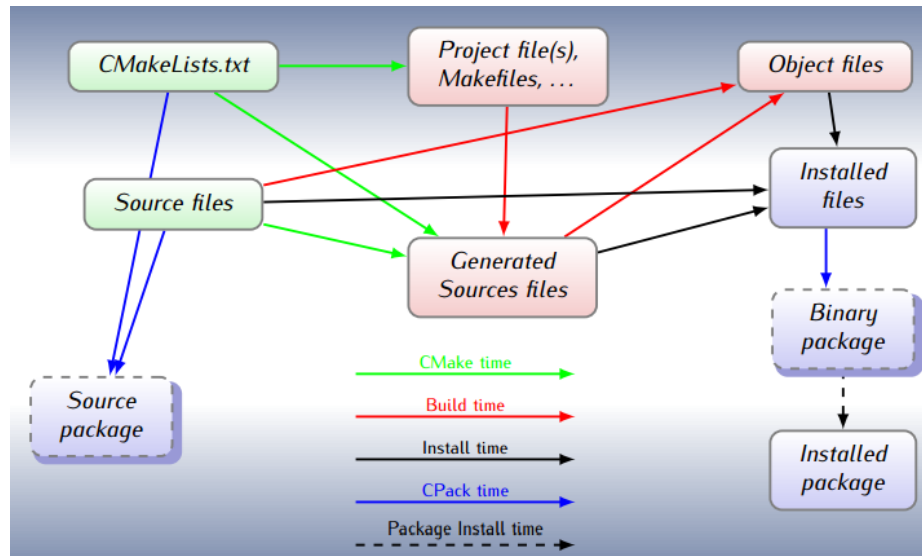
CMake



- В настоящий момент является стандартом де-факто для C++ проектов
- Максимальная свобода в выборе окружения разработки (в рамках одной команды!)
- Широкая поддержка разнообразных целевых платформ

CMake Workflow

CMakeLists.txt — файл, описывающий порядок сборки приложения



- Шаг 1. Генерация *проектных файлов* при помощи cmake или CMakeGui
 - *.vcproj, Makefile, etc*
- Шаг 2. Компиляция исходников при помощи компиляторов из Visual Studio, Qt Creator, Eclipse, XCode...
 - *.obj, .o*
- Шаг 3. Линковка финальных бинарных файлов компоновщиком (link.exe, ld, ...)
 - *.exe, .dll, .lib, .a, .so, .dylib*

Пример сборки

Содержимое каталога:

```
code
├─ CMakeLists.txt
├─ lib.h
├─ lib.c
└─ main.c
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)
project(first_sample)

set(SOURCES main.c lib.c)
add_executable(sample_app ${SOURCES}) # Объявляет исполняемый модуль с именем sample_app
```

Построение вне дерева с исходниками

Плохо: в директории с исходным кодом

```
code
├─ hello.hpp
├─ hello.cpp
└─ hello.exe
```

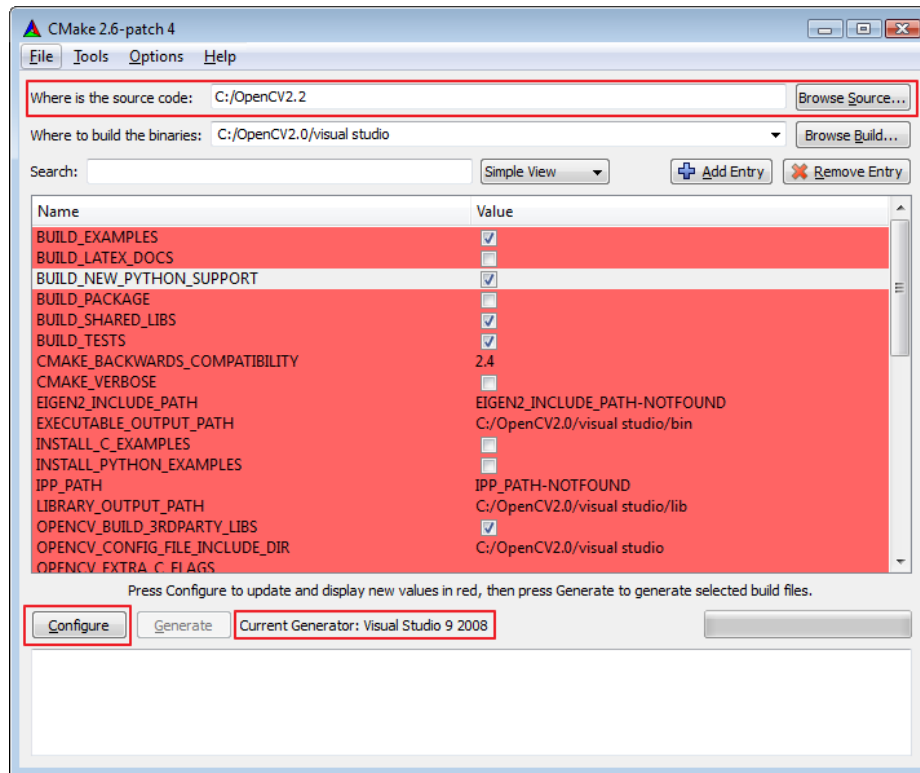
Хорошо: вне директории (чистый репозиторий, несколько build-директорий)

```
code
├─ hello.hpp
├─ hello.cpp
build
└─ hello.exe
```

Соответствующие команды:

```
$ cd <code>
$ mkdir ../build
$ cd ../build
$ cmake ../code
$ make
```

CMake GUI



Debug / Release

В CMakeLists.txt:

```
SET(CMAKE_BUILD_TYPE Debug)
```

В командной строке:

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ../code # Запомните эту команду!
```

Для библиотек:

```
TARGET_LINK_LIBRARIES(lib RELEASE ${lib_SRCS})  
TARGET_LINK_LIBRARIES(libd DEBUG ${lib_SRCS})
```

Пример сборки библиотеки

Содержимое каталога:

```
code
├─ CMakeLists.txt
├─ lib.h
├─ lib.c
└─ main.c
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)
project(second_sample)

set(SOURCE_LIB lib.c)
add_library(library STATIC ${SOURCE_LIB}) # Объявляет библиотеку с именем library

set(SOURCES main.c)
add_executable(main ${SOURCES}) # Объявляет исполняемый модуль с именем sample_app
target_link_libraries(sample_app library) # Указывает зависимость от библиотеки
```

Добавление подпроекта

Содержимое каталога:

```
code
├── CMakeLists.txt
├── library
│   ├── CMakeLists.txt
│   ├── lib.c
│   └── lib.h
└── main.c
```

Корневой CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)
project(third_sample)

add_subdirectory(library) # Указывает, что в директории library есть свой CMakeLists.txt

include_directories(library)
set(SOURCES main.c)
add_executable(sample_app ${SOURCES})

target_link_libraries(sample_app library)
```

library/CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)
project(library)

set(SOURCE_LIB lib.c)
add_library(library STATIC ${SOURCE_LIB})
```

Поиск зависимостей

CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)
project(sample)

# Поиск OpenCV
find_package(OPENCV REQUIRED)
if(NOT OPENCV_FOUND)
    message(SEND_ERROR "Failed to find OpenCV")
    return()
else()
    include_directories(${OPENCV_INCLUDE_DIR})
endif()

add_executable(sample_app main.c)
target_link_libraries(sample_app ${OPENCV_LIBRARIES})
```


CMake резюме

- Поначалу нетривиален, но очень удобен впоследствии
- Дает членам команды максимальную свободу в выборе инструментов
- Является стандартом де-факто для кросс-платформенных C++ проектов



Коллективная работа с кодом

1. Необходимо центральное хранилище кода

- *Актуальное и используемое всеми участниками (где последняя версия?!)*
- *Защищенное, с разграничением прав доступа*

2. Нужно иметь возможность просматривать историю изменений

- *Откат дефектных изменений*
- *Извлечение кода "из прошлого" (как оно раньше работало?)*
- *Поиск ошибок сравнением (кто это сделал?)*

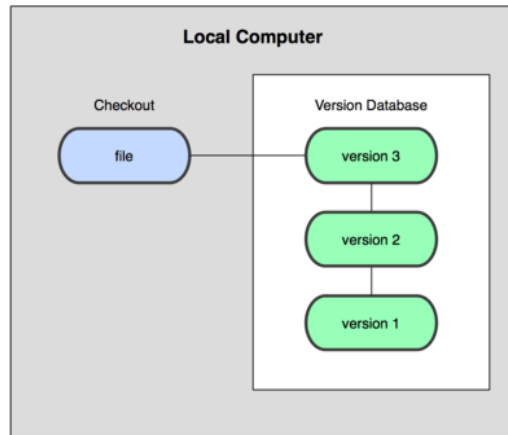
Нужны ли специальные инструменты?

Системы контроля версий

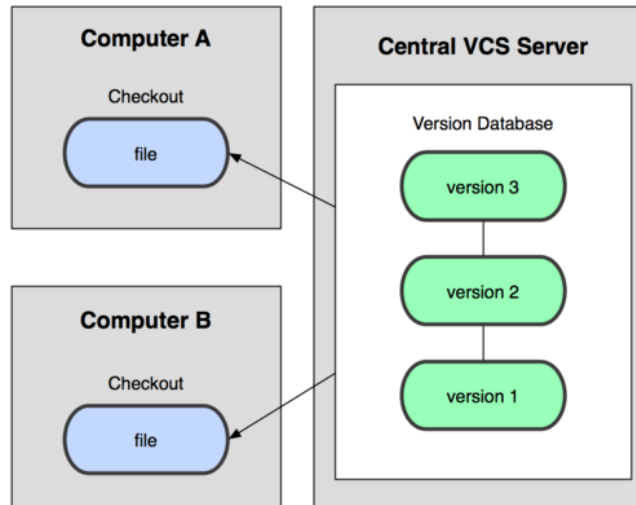
Системы контроля версий — это программные системы, хранящие несколько версий одного документа, и позволяющие вернуться к более ранним версиям. Как правило, для каждого изменения запоминается дата модификации и автор.

- Обычно используются для хранения исходного кода (source control), но имеются и другие применения: конфигурации, документация, компьютерная анимация, САПР и др.
- Являются одним из важнейших инструментов разработки, появляется все больше примеров использования в других отраслях ([книгоиздание](#), [официальные документы](#)).

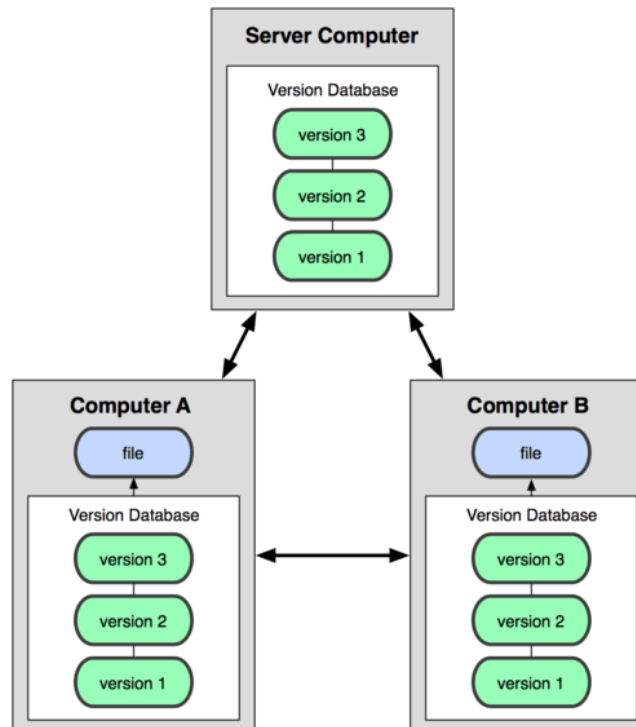
Три поколения VCS: Локальные



Три поколения VCS: Централизованные



Три поколения VCS: Распределенные

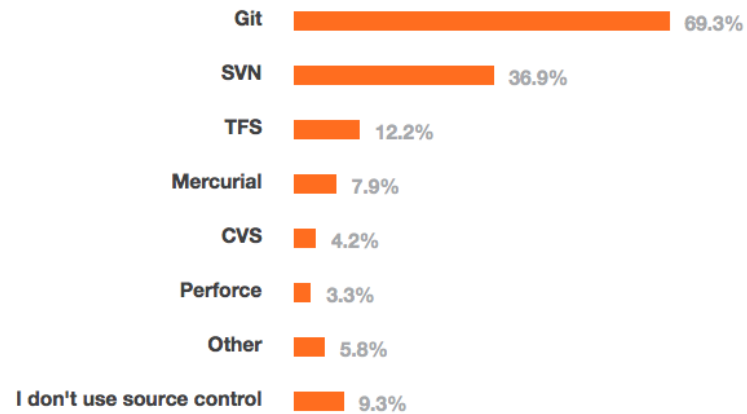


Три поколения VCS

Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe, Subversion, Team Foundation Server
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

Eric Sink ["A History of Version Control"](#)

Популярные VCS



16,694 responses

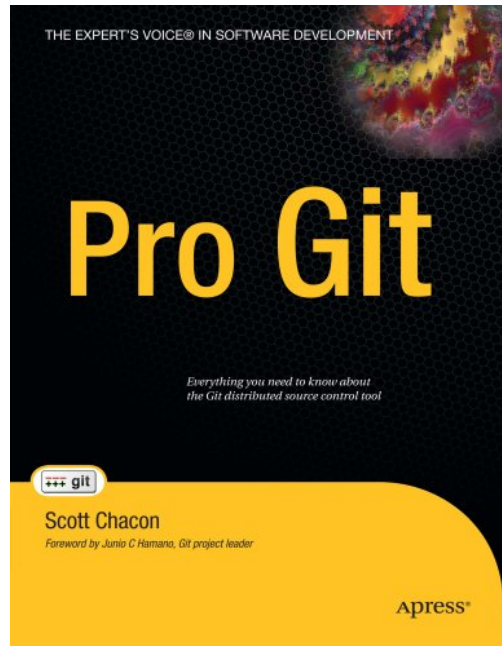
Stack Overflow Developer Survey 2015

Git



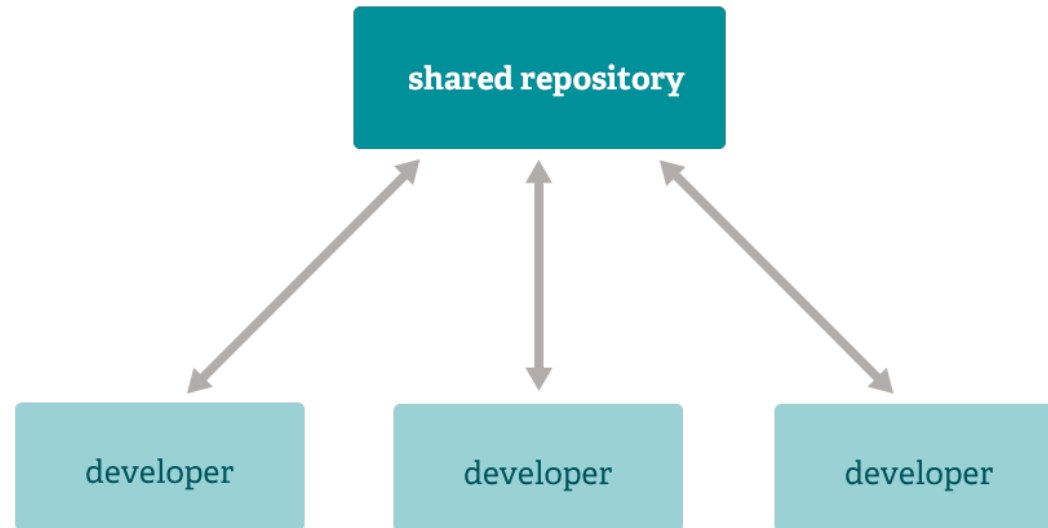
- Изначально разработан Линусом Торвальдсом для работы над ядром Linux.
- В настоящее время поддерживается Джунио Хамано, сотрудником Google.
- Не очень прост в освоении, однако очень быстрый и функциональный.
- Главное преимущество — свобода выбора рабочего процесса (workflow).
- Имеет наиболее "сильное" сообщество, инструментальную поддержку.
- Официальный сайт проекта: <http://www.git-scm.org>.

Pro Git



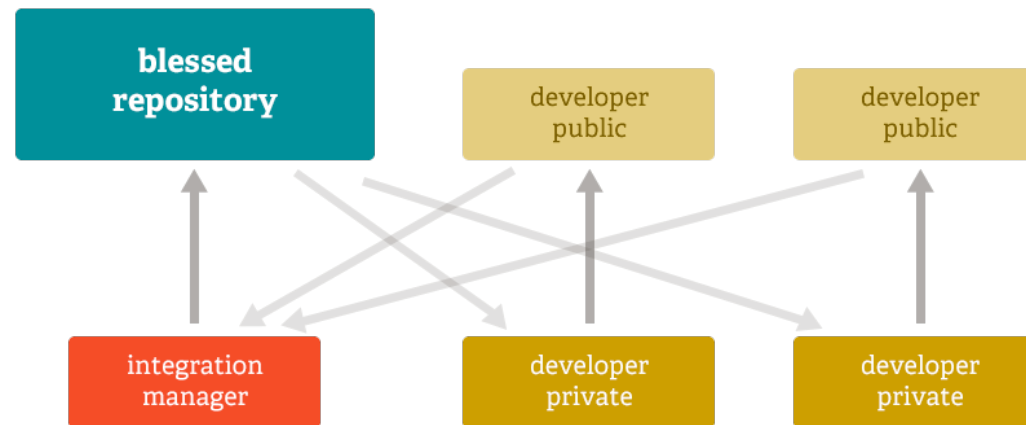
- Лучшая книга про Git
- Доступна бесплатно
- Переведена на [русский язык](#)
- Единственный способ по-настоящему понять Git — это узнать как он работает
- Нужно прочесть хотя бы первые 100 страниц

Centralized Workflow



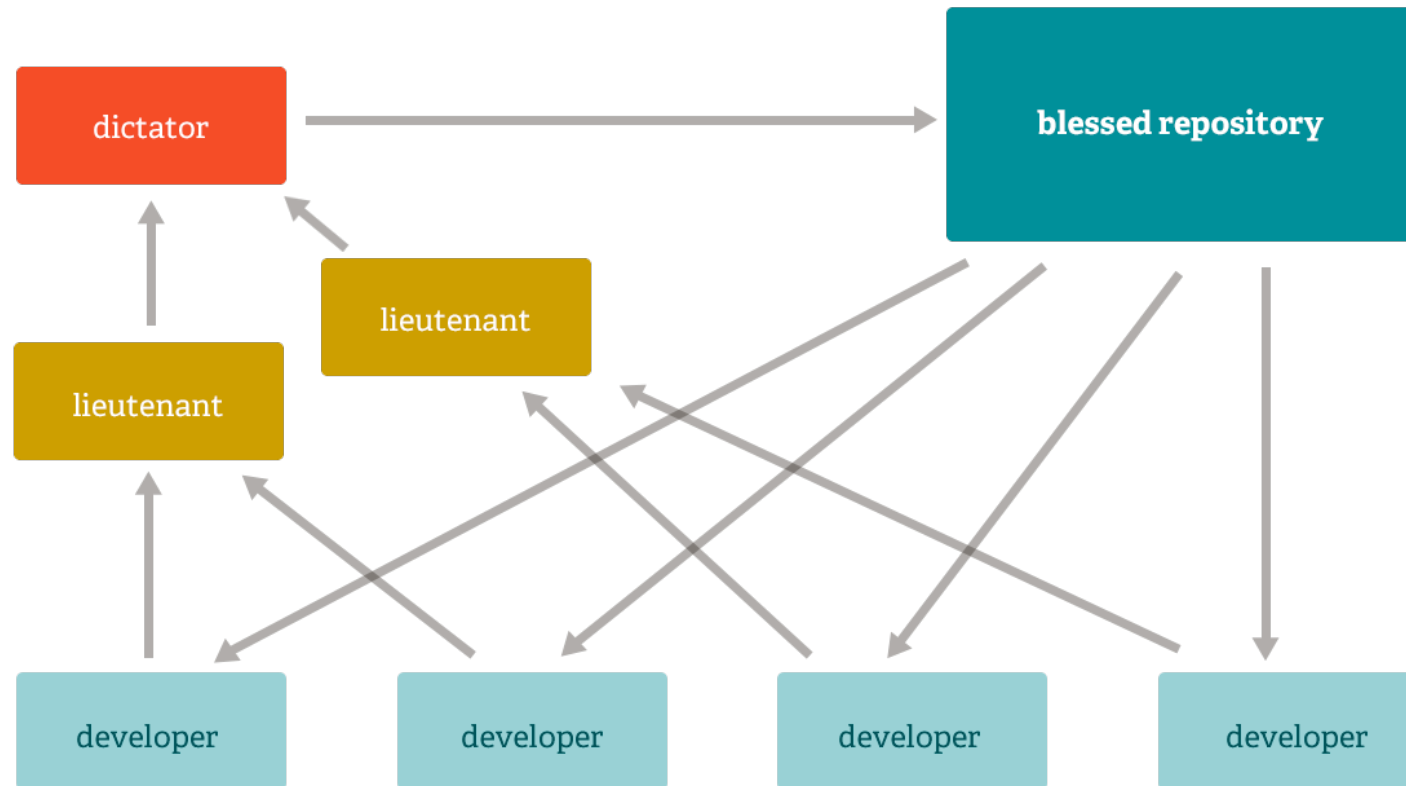
Каковы достоинства и недостатки данного подхода?

Integration Manager Workflow



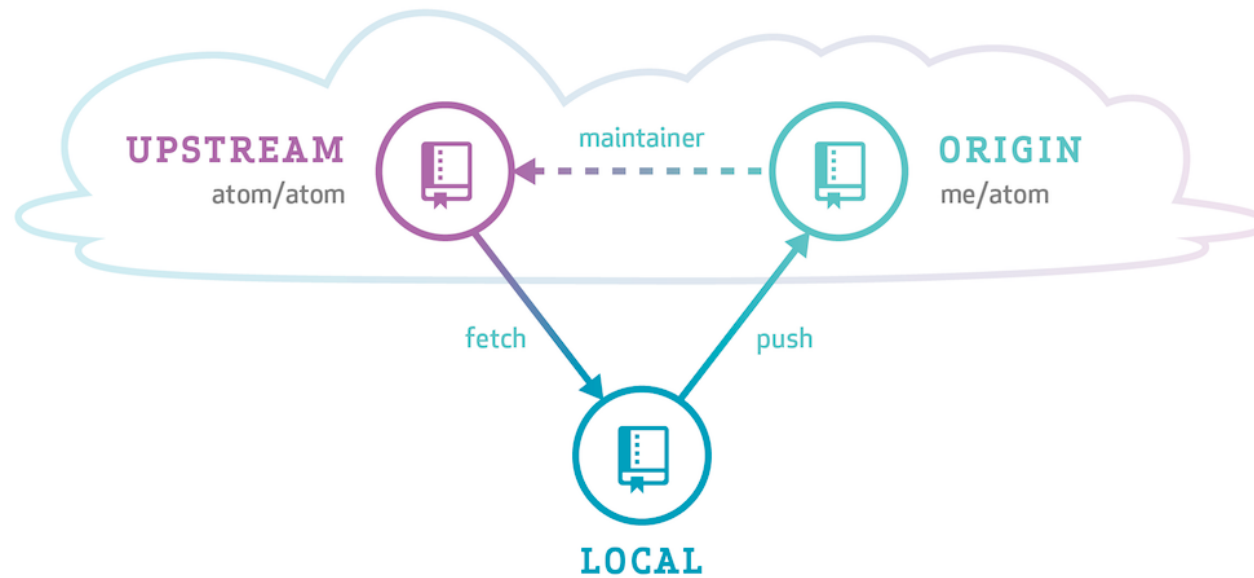
Каковы достоинства и недостатки данного подхода?

Dictator and Lieutenants Workflow



Каковы достоинства и недостатки данного подхода?

Triangular Workflow (GitHub)



```
$ cd practice1-devtools
$ git remote -v
origin https://github.com/kirill-kornyakov/practice1-devtools.git (fetch)
origin https://github.com/kirill-kornyakov/practice1-devtools.git (push)
upstream https://github.com/Itseez-NNSU-SummerSchool2015/practice1-devtools.git (fetch)
upstream https://github.com/Itseez-NNSU-SummerSchool2015/practice1-devtools.git (push)
```

Основные термины

- repository, depot
- working copy
- revision
- head
- check-out, clone
- update, sync
- check-in, commit, submit
- commit, changeset, patch
- pull/merge request
- merge, integration
- conflict
- rebase
- shelving, stashing
- branch
- trunk, mainline, master
- tag, label

[Глоссарий](#)

Базовые принципы

1. Стабильность общих (публичных) веток:

- *Они обязаны компилироваться и проходить все тесты в любой момент времени.*
- *Изменения тестируются до попадания в репозиторий.*
- *Если дефектные изменения прошли, они исправляются в срочном порядке.*


2. Абсолютно вся разработка фиксируется в истории:



- *Это делается в виде отдельных веток локального или глобального репозитория.*
- *"Удачные" изменения добавляются в основную ветвь.*

Git Commits


GitHub



Commits on Aug 15, 2015

**Merge pull request #12 from kirill-kornyakov/minor-cleanings** ...


 9fc4813 



kirill-kornyakov authored 8 hours ago

**Merge pull request #14 from valentina-kustikova/master** ...


 f47b7d8 



kirill-kornyakov authored 8 hours ago

**Updated description.**

 13622a8 

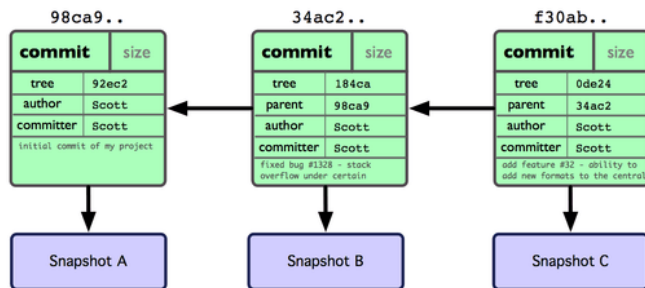
valentina-kustikova authored 9 hours ago

**Readme modifications.**

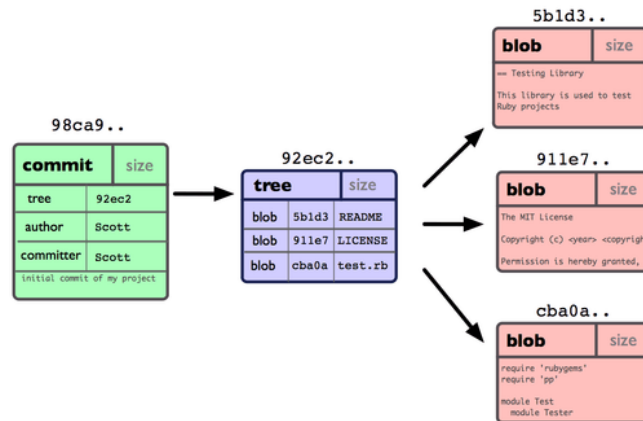
 7630f49 

valentina-kustikova authored 10 hours ago

Git

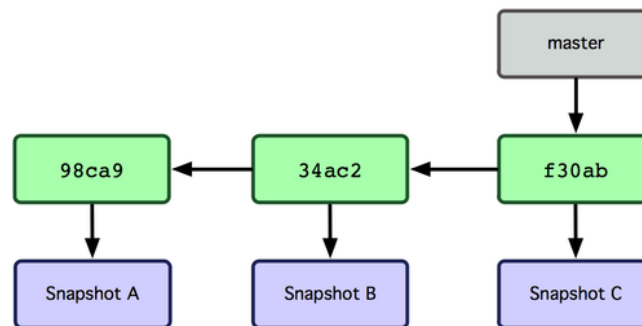


Git Objects



Branches

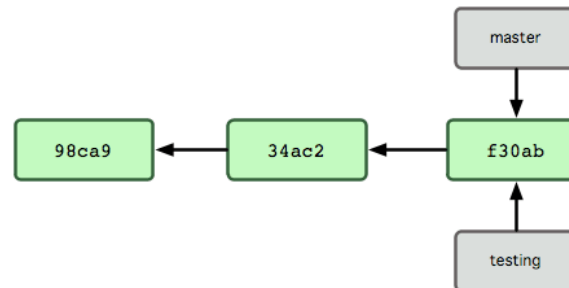
- Ветки в Git — это просто указатели на коммиты
- `master` — это общепринятое название для ветки, указывающей на актуальное состояние репозитория



git branch

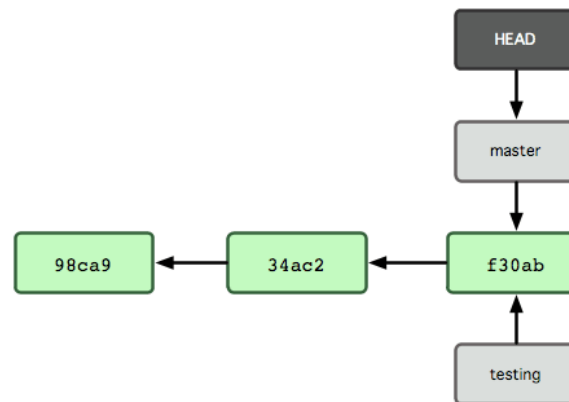
```
$ git branch testing
```

- Создание новой ветки — это всего лишь создание нового указателя
- Копирования файлов при этом не происходит



HEAD

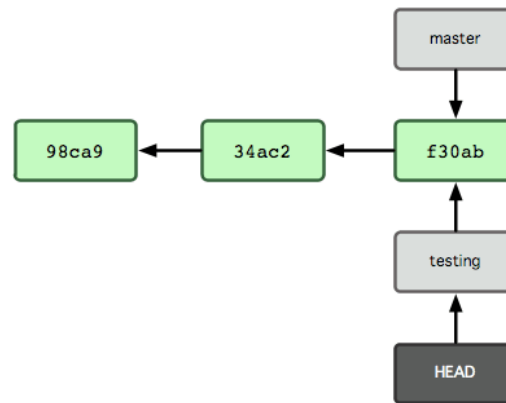
- HEAD — это указатель на текущую ветку
- Он обновляется автоматически в зависимости от выполненных команд



git checkout

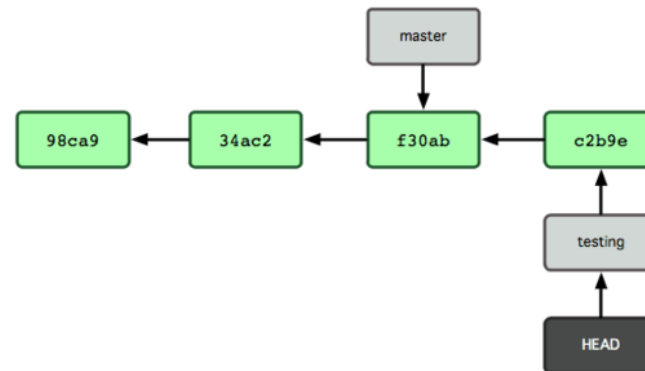
```
$ git checkout testing
```

- Указатель HEAD автоматически переместился на выбранную ветку



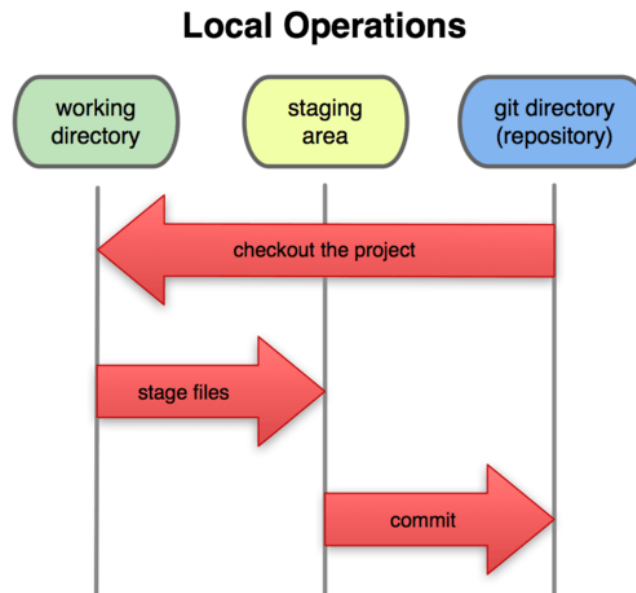
git commit

```
$ vim README.md  
# Edit the README.md file  
$ git add README.md  
$ git commit -m 'Added information to the README'
```



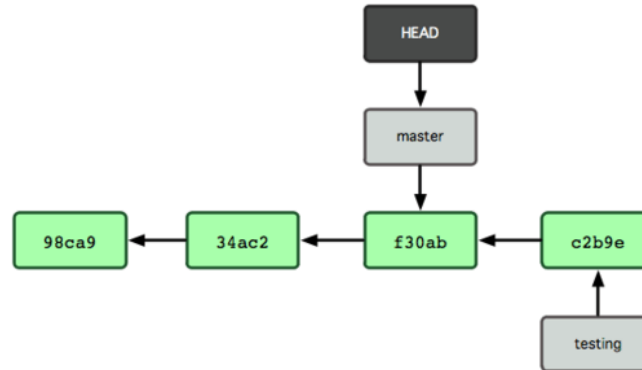
- Обратите внимание, что `testing` и `HEAD` автоматически передвинулись.
- При этом `master` остается на месте, поскольку мы работаем в ветке `testing`.

Три состояния файлов



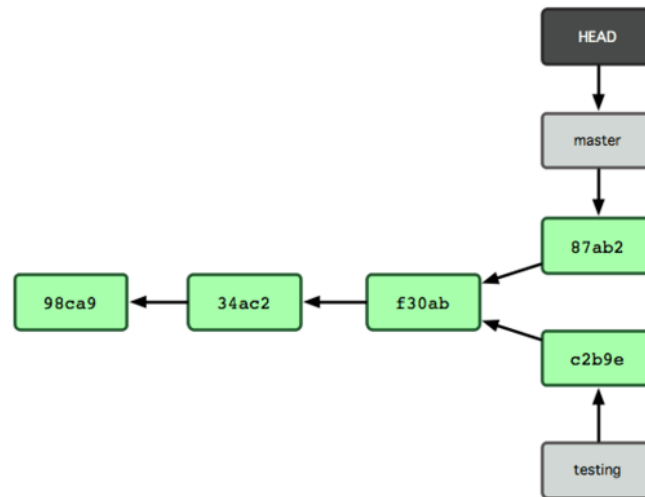
Go back to master

```
$ git checkout master
```

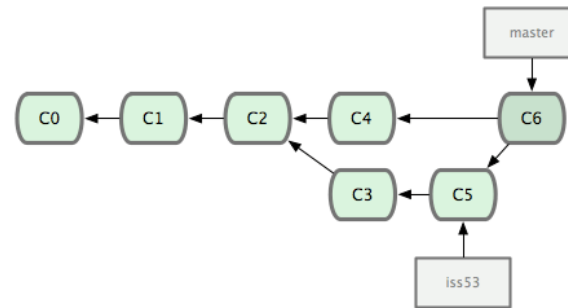
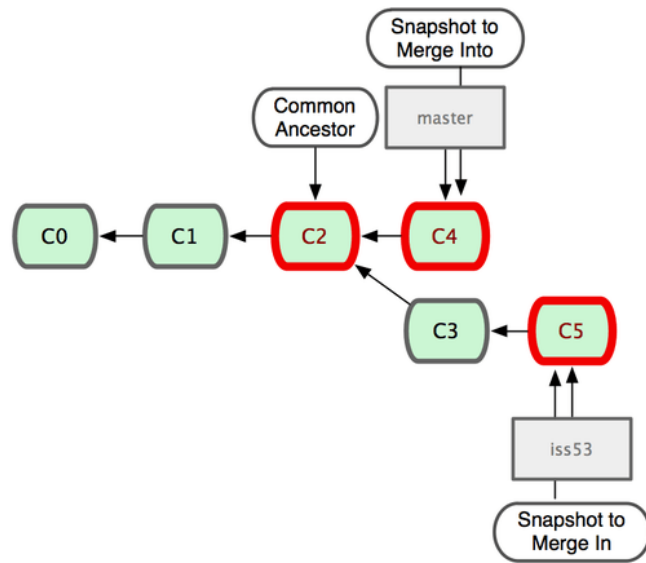


Make a commit to master

```
$ vim main.cpp  
$ git commit -a -m 'made other changes'
```

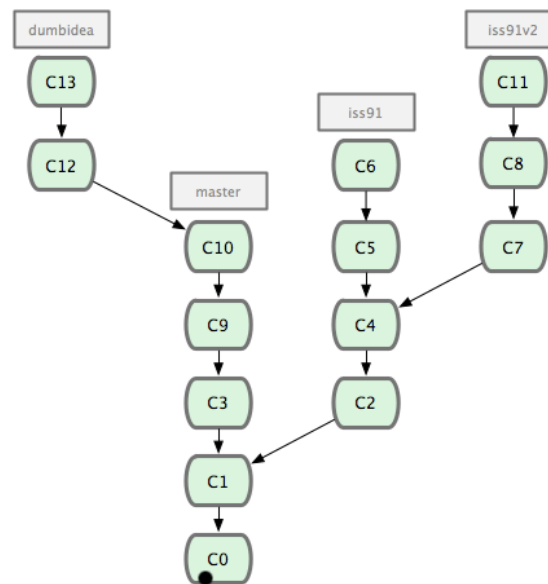


Merging



Multiple branches

- Хорошим тоном считается создание отдельной ветки для каждого логически независимого изменения
- У одного человека может быть десяток активных веток



GitHub Flow

github / github

Admin Unwatch Fork Your Fork Pull Request 16 12

Source Commits Network Pull Requests (23) Fork Queue Issues (290) Wiki (98) Graphs Branch: master

Switch Branches (139) Switch Tags (0) Branch List Search source code...

Branches

Showing 30 of 139 branches

Recently Active Stale

master	Base branch
Last updated about 5 hours ago by tekub	
fi-signup Last updated 36 minutes ago by sr	3 ahead 0 behind
charlock-linguist Last updated about 13 hours ago by josh	11 ahead 7 behind
git-http-server Last updated about 14 hours ago by romayko	11 ahead 7 behind
wild-renaming Last updated about 20 hours ago by defunkt	3 ahead 25 behind
no-inline-js-config Last updated 1 day ago by josh	108 ahead 37 behind
svg-tests Last updated 1 day ago by jsncostello	2 ahead 45 behind
knyle-style-commits Last updated 1 day ago by kneath	40 ahead 73 behind
enterprise-non-config Last updated 2 days ago by romayko	7 ahead 64 behind
menu-behavior-act-i Last updated 4 days ago by josh	5 ahead 150 behind
view-modes Last updated 5 days ago by kneath	36 ahead 209 behind

GitHub Flow

GitHub Flow

Anything in the `master` branch is deployable.

1. Create branch

- *To work on something new, create a descriptively named branch off of `master` (ie: `new-oauth2-scopes`).*

2. Develop in branch

- *Commit to that branch locally and regularly push your work to the same named branch on the server.*

3. Open a pull request (ask for review)

- *When you need feedback or help, or you think the branch is ready for merging, open a pull request.*

4. Merge after review

- *After someone else has reviewed and signed off on the feature, you can merge it into `master`.*

5. Deploy

- *Once it is merged and pushed to `master`, you can and should deploy immediately.*

GitHub Flow

```
# Check that origin and upstream repositories are correctly defined
$ git remote -v

# Get the latest sources from the upstream repository
$ git remote update

# Checkout a new topic branch for development
$ git checkout -b adding-new-feature upstream/master

#
# Do some development...
#

# Check your changes
$ git status

# Commit your changes
$ git commit -a -m "Added a new feature"

# Push your changes to the origin
$ git push origin HEAD
```




Фреймворки для Unit-тестирования

Значительно упрощают создание и запуск unit-тестов, позволяют придерживаться единого стиля.

1. xUnit — общее обозначение для подобных фреймворков.

2. Бесплатно доступны для большинства языков:

- *C/C++: CUnit, CPPUnit, GoogleTest*
- *Java: JUnit*
- *.NET: NUnit*

3. Встроены в некоторые языки:

- *D, Python, Go*

Типичные возможности

1. Удобное добавление тестов

- Простая регистрация новых тестов
- Набор функций-проверок (*assert*)
- Общие инициализации и деинициализации

2. Удобный запуск тестов

- Пакетный режим
- Возможность фильтрации тестов по именам

3. Допускают интеграцию с IDE

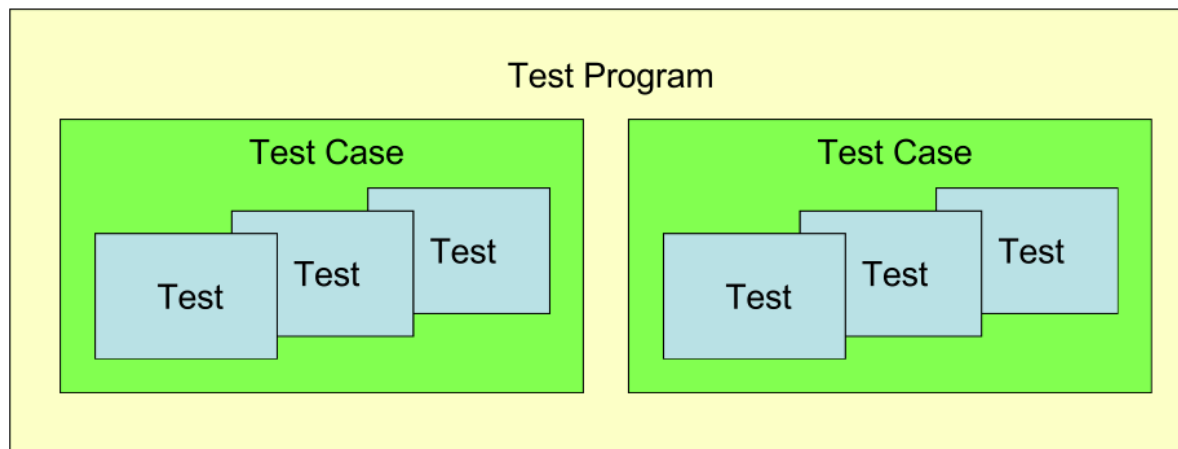
4. Генерация отчета в стандартном XML-формате

- Возможность последующего автоматического анализа
- Публикация на web-страницах проекта

Google Test

1. Популярный фреймворк для написания модульных тестов на C++, разработанный Google.
2. Используется в целом ряде крупных проектов
 - *Chromium, LLVM компилятор, OpenCV*
3. Написан на C++, строится при помощи CMake
 - *Поддерживает: Linux, Mac OS X, Windows, Cygwin, Windows CE и Symbian*
4. **Open-source** проект с BSD лицензией (допускает использование в закрытых коммерческих проектах).
5. Как правило используется в консольном режиме, но существует вспомогательное GUI **приложение**.

Базовые концепции



- Совокупность тестов образует "обычное" консольное приложение.
- Каждый тест представляет собой "обычную" функцию, объявленную с использованием макроса `TEST ()` или `TEST_F ()`.
- `TEST ()` не только определяет, но и "регистрирует" тест.

Пример 1

```
#include <gtest/gtest.h>

TEST(MathTest, two_plus_two_equals_four)
{
    int x = 2 + 2;

    EXPECT_EQ(4, x);
}

TEST(MathTest, two_not_equal_three)
{
    int x = 2;
    int y = 3;

    EXPECT_NE(x, y);
}
```

Пример 2

Функция

```
int Factorial(int n); // Returns the factorial of n
```

Тесты

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(2, Factorial(2));
    EXPECT_EQ(6, Factorial(3));
    EXPECT_EQ(40320, Factorial(8));
}
```

Пример 3

```
#include <gtest/gtest.h>
#include <vector>

using namespace std;

// A new one of these is created for each test
class VectorTest : public testing::Test {
public:
    vector<int> m_vector;

    virtual void SetUp() {
        m_vector.push_back(1);
        m_vector.push_back(2);
    }

    virtual void TearDown() {}
};

TEST_F(VectorTest, testElementZeroIsOne) {
    EXPECT_EQ(m_vector[0], 1);
}

TEST_F(VectorTest, testElementOneIsTwo) {
    EXPECT_EQ(m_vector[1], 2);
}

TEST_F(VectorTest, testSizeIsTwo) {
    EXPECT_EQ(m_vector.size(), (unsigned int)2);
}
```


Консольный лог Google Test

```
$ bin/hellotest

Running main() from gtest_main.cc
[=====] Running 4 tests from 2 test cases.
[-----] Global test environment set-up.
[-----] 3 tests from VectorTest
[ RUN      ] VectorTest.testElementZeroIsOne
[          OK ] VectorTest.testElementZeroIsOne (0 ms)
[ RUN      ] VectorTest.testElementOneIsTwo
[          OK ] VectorTest.testElementOneIsTwo (0 ms)
[ RUN      ] VectorTest.testSizeIsTwo
[          OK ] VectorTest.testSizeIsTwo (0 ms)
[-----] 3 tests from VectorTest (0 ms total)

[-----] 1 test from MathTest
[ RUN      ] MathTest.Zero
[          OK ] MathTest.Zero (0 ms)
[-----] 1 test from MathTest (0 ms total)

[-----] Global test environment tear-down
[=====] 4 tests from 2 test cases ran. (0 ms total)
[ PASSED   ] 4 tests.
```

Полезные советы

Тесты можно временно выключать

```
TEST(MathTest, DISABLED_two_plus_two_equals_four)
{
    int x = 2 + 2;

    EXPECT_EQ(4, x);
}
```

Тесты можно фильтровать по имени при запуске

```
$ ./bin/hellotest --gtest_filter=*Vector*
```

У Google Test есть ряд других полезных опций

```
$ ./bin/hellotest --help
```

Юнит-тест курильщика

```
[Test]
public void TestMethod1()
{
    var calc = new Calculator();
    calc.ValidOperation = Calculator.Operation.Multiply;
    calc.ValidType = typeof(int);
    var result = calc.Multiply(-1, 3);
    Assert.AreEqual(result, -3);
    calc.ValidOperation = Calculator.Operation.Multiply;
    calc.ValidType = typeof(int);
    result = calc.Multiply(1, 3);
    Assert.IsTrue(result == 3);
    if (calc.ValidOperation == Calculator.Operation.Invalid)
    {
        throw new Exception("Operation should be valid");
    }
    calc.ValidOperation = Calculator.Operation.Multiply;
    calc.ValidType = typeof(int);
    result = calc.Multiply(10, 3);
    Assert.AreEqual(result, 30);
}
```

Авторство: Антон Бевзюк, SmartStepGroup.

Юнит-тест здорового человека

```
[TestMethod]
public void CanAuthenticateUser() {
    var page = new TestableLoginPage();

    page.AuthenticateUser("user", "user");

    Assert.AreEqual("user", page.AuthenticatedUser);
}
```


Критерии хорошего теста

1. Короткий, понятный, имеет чистый код
2. Сфокусированный (только один assert)
3. Быстрый
4. Автоматический
5. Независим от порядка исполнения и окружения

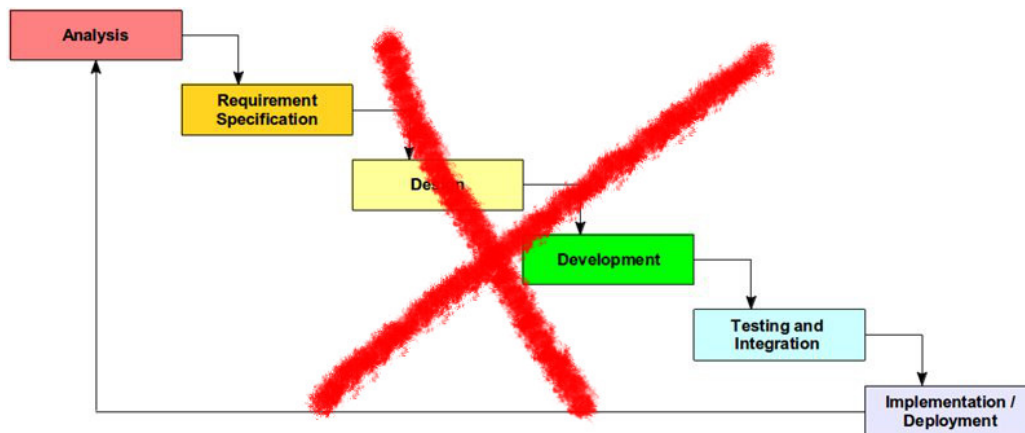


TRAVIS

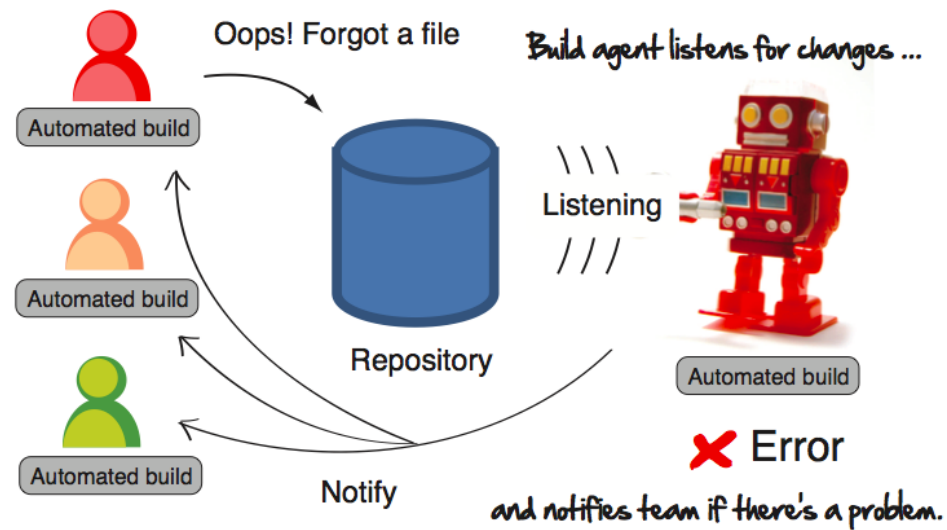
Непрерывная интеграция

*Непрерывная интеграция (англ. Continuous Integration) – это практика разработки ПО, которая заключается в выполнении **частых автоматизированных сборок** проекта для скорейшего выявления и решения интеграционных проблем.*

Непрерывная сборка – это сердцебиение вашего проекта.



Практика непрерывной интеграции



1. Все хранится в **централизованном репозитории**:
исходный код, конфигурационные файлы, тестовые данные, сборочные и тестовые скрипты
2. **Полная автоматизация** операций с кодом:
выкачивание из репозитория, сборка, тестирование и так далее.

Travis CI



- Официальный сайт проекта: <http://travis-ci.org>
- Веб-сервис для сборки и тестирования ПО ([open-source](#))
- Важными особенностями являются интеграция с GitHub и возможность бесплатного использования
- Поддерживает большое количество языков: C, C++, Clojure, Erlang, Go, Groovy, Haskell, Java, JavaScript, Perl, PHP, Python, Ruby и Scala
- Тестирование происходит на виртуальных Linux-машинах, запускаемых в облаке Amazon

GitHub + Travis CI

Conversation 5 Commits 7 Files changed 20

Commits on Mar 7, 2015

- Added lab3**
Nikolay-Borisov authored on 7 Mar ✖
- copy/paste fixes**
Nikolay-Borisov authored on 7 Mar ✖
- copy/paste fixes 2**
Nikolay-Borisov authored on 7 Mar ✖
- copy/paste fixes 3**
Nikolay-Borisov authored on 7 Mar ✔
- added ViewModelWithTxtLoggerTests.java**
Nikolay-Borisov authored on 7 Mar ✖
- super commit**
Nikolay-Borisov authored on 7 Mar ✔

Commits on Mar 9, 2015

- reformat code**
Nikolay-Borisov authored on 9 Mar ✔

UNN-VMK-Software/agile-course-practice build passing

Current	Branches	Build History	Pull Requests
✔	PR #176 Борисов - Лабораторная работа #3	# 1477 passed	
🔗	⦿ Borisov-Nikolay committed	👤 28f5291	
✔	PR #176 Борисов - Лабораторная работа #3	# 1476 passed	
🔗	⦿ Borisov-Nikolay committed	👤 8a5db86	
✖	PR #176 Борисов - Лабораторная работа #3	# 1475 failed	
🔗	⦿ Borisov-Nikolay committed	👤 928f34d	
✔	PR #176 Борисов - Лабораторная работа #3	# 1474 passed	
🔗	⦿ Borisov-Nikolay committed	👤 6775790	
✖	PR #176 Борисов - Лабораторная работа #3	# 1473 failed	
🔗	⦿ Borisov-Nikolay committed	👤 3356825	
✖	PR #176 Борисов - Лабораторная работа #3	# 1472 failed	
🔗	⦿ Borisov-Nikolay committed	👤 917a45d	

Современная стратегия тестирования

- Без "зеленых" тестов нет уверенности в работоспособности кода
- Фокус на максимальную автоматизацию
 - *Полное тестирование требуется несколько раз в день, каждому члену команды*
- Тесты пишутся самими разработчиками, одновременно с реализацией
 - *Тесты это лучшая документация, которая всегда актуальна (компилятор!)*
 - *Тесты это первые сэмплы, показывающие простые примеры использования*
 - *Test-Driven Development*
- Код тестируется **непрерывно**
 - *Это делается локально на машине разработчика*
 - *Это делается на сервере до того, как добавить его в репозиторий*

Современная стратегия тестирования (2)

- Автоматические тесты замещают отладку
 - *Предсказуемость времени разработки*
 - *Пойманный баг документируется в виде теста*
- Тесты — это "first-class citizens"
 - *Стоит отдавать код вместе с тестами*
 - *Нужно заботиться о качестве кода тестов*
 - *Метафора тестов: скелет, позволяющий организму двигаться*

Резюме

В рамках нашей школы будут выполняться *учебные* задания, однако они будут носить все черты *промышленной* разработки ПО.

■ Кросс-платформенность

- *Построение с использованием CMake*
- *Тестирование на Linux (gcc, clang)*
- *Потенциальная переносимость на Android и iOS*

■ Коллективная разработка

- *Использование Git*
- *Peer code review от преподавателей и коллег средствами GitHub*

■ Тестирование

- *Автоматические тесты на базе Google Test*
- *Непрерывная интеграция на основе Travis-CI*
- *Автоматический анализ качества алгоритмов с использованием Python*

Ссылки

1. Wikipedia ["Системы контроля версий"](#).
2. [Pro Git](#) by Scott Chacon.
3. ["Mercurial tutorial"](#) by Joel Spolsky.
4. [GTest](#)
5. [Google Test Talk](#)

Спасибо!

Вопросы?