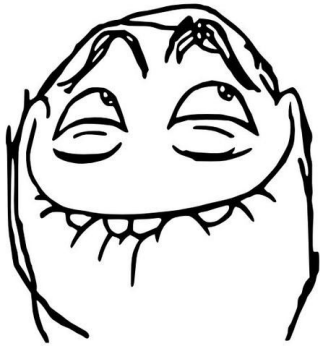




# Функциональное программирование на языке Scala

**Tinkoff.ru**

# Введение в тестирование





А для чего вообще нужно тестирование разработчику?

- проверить, что написанный код работает
  - успешные сценарии
  - неуспешные сценарии
  - граничные условия
- защитить код от поломок при рефакторинге
- сформулировать требования к коду перед его написанием
- убедиться, что ничего не забыто после написания кода



## Test Driven Development (TDD)

### 1. Сначала пишем тесты

- приходится думать о декомпозиции задачи на сцепленные слабо зависимые куски
- формализуем сценарии использования и требования к кускам кода **до изменения кода**
- новые тесты не должны проходить

### 2. Пишем код, выполняющий эти тесты

- не пишем код, не покрытый тестами

### 3. Рефакторим

### 4. Повторяем, пока не надоест (до 10 изменений за итерацию)

# Введение в тестирование: assert



```
def assert(assertion: Boolean) {  
    if (!assertion)  
        throw new java.lang.AssertionError("assertion failed")  
}  
  
def assert(assertion: Boolean, message: => Any) {  
    if (!assertion)  
        throw new java.lang.AssertionError("assertion failed: "+ message)  
}
```

```
val a = 2 + 2 * 2  
assert(a == 6, "Something is wrong with your math")
```

```
# Enable/disable assertions in command-line:  
java [ -ea | -da ] <class name>
```



ScalaTest - универсальный комбайн юнит-тестирования в Scala.

- Suite - набор из 0+ тестов
- Тест - это именованное нечто, что можно запустить и получить один из следующих результатов: успех, провал, отложено, отменено
- Трейт Suite определяет базовые методы для работы с набором тестов
- Разные наследники Suite реализуют разные стили тестирования

# Введение в тестирование: ScalaTest FunSuite



```
import org.scalatest.FunSuite

class SetSuite extends FunSuite {
  test("An empty Set should have size 0") {
    assert(Set.empty.size == 0)
    assertResult(0) { Set.empty.size }
  }

  test("Invoking head on an empty Set should produce NoSuchElementException") {
    assertThrows[NoSuchElementException] {
      Set.empty.head
    }
  }
}
```

```
protected def test(testName: String, testTags: Tag*)
  (testFun: => Any)
  (implicit pos: source.Position): Unit
```



## BDD – Behavior-Driven Development

- Суть: ЧПУ ЧПОТ – Человекопонятные определения тестов
- Сначала пишем, что мы подвергаем проверкам (субъект)
- Потом пишем, что и как этот субъект должен делать (поведение)
- Матчеры предоставляют человекопонятный DSL для проверок
- Есть ещё FeatureSpec, WordSpec и другие умные слова, но о них позже



# Введение в тестирование: ScalaTest BDD



```
import org.scalatest.{FlatSpec, Matchers}
import Element.elem

class ElementSpec extends FlatSpec with Matchers {
  "A UniformElement" should "have a width equal to the passed value" in {
    val ele = elem('x', 2, 3)
    ele.width should be(2)
  }
  it should "have a height equal to the passed value" in {
    val ele = elem('x', 2, 3)
    ele.height should be(3)
  }
  it should "throw an IAE if passed a negative width" in {
    an[IllegalArgumentException] should be thrownBy {
      elem('x', -2, 3)
    }
  }
}
```

# Введение в тестирование: Домашнее задание

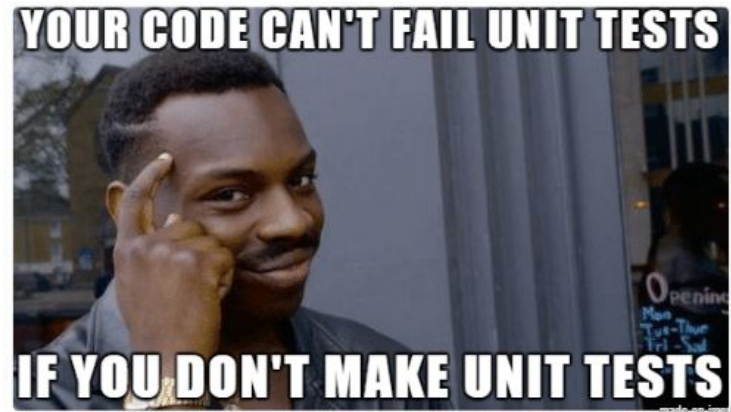


Реализуйте тесты для следующих задач:

- `lectures.functions.Fibonacci`
- `lectures.functions.Fibonacci2`
- `lectures.functions.Computation`
- `lectures.functions.CurriedComputation`,
- `lectures.functions.FunctionalComputation`
- `lectures.functions.SQLAPI`
- `lectures.operators.Competition`

Для всех задач необходимо проверить основные успешные сценарии.

Для двух задач про числа Фибоначчи - еще и неуспешные сценарии и граничные условия. В случае неуспешного сценария должно бросаться исключение `RuntimeException` с понятным текстом ошибки.



# Среда разработки и тестирования



## Основные задачи IDE

- Подсветка синтаксиса и удобная навигация по коду?

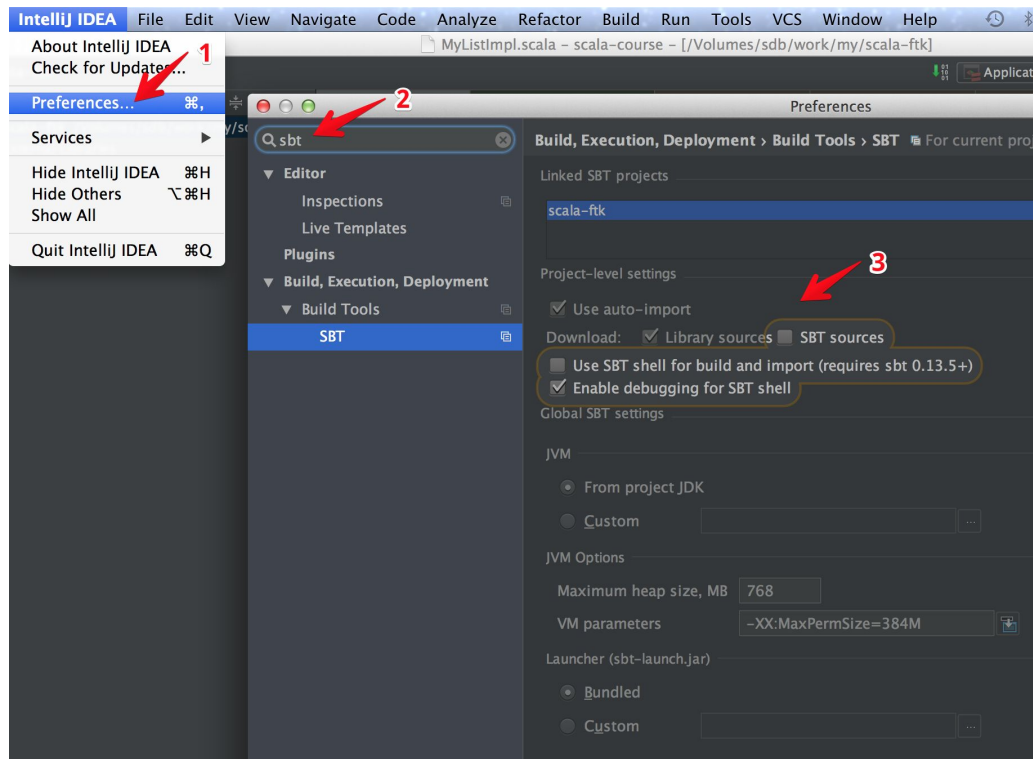


IDE - кухонный комбайн по изготовлению программ и доставке их до прода:

- Еще более удобное редактирование: подсказки, автодополнения, проверка синтаксиса, поддержка фреймворков
- Компиляция
- Отладка
- Юнит-тестирование
- Работа с БД
- Работа с системами контроля версий



## Настройка, установка плагинов



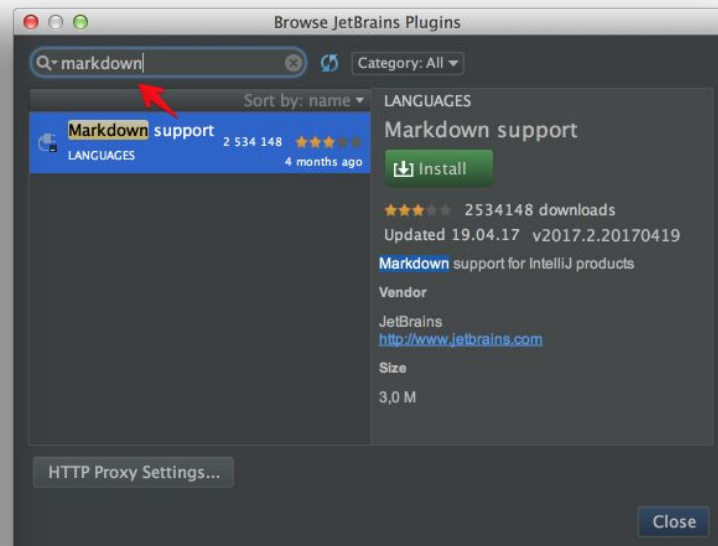
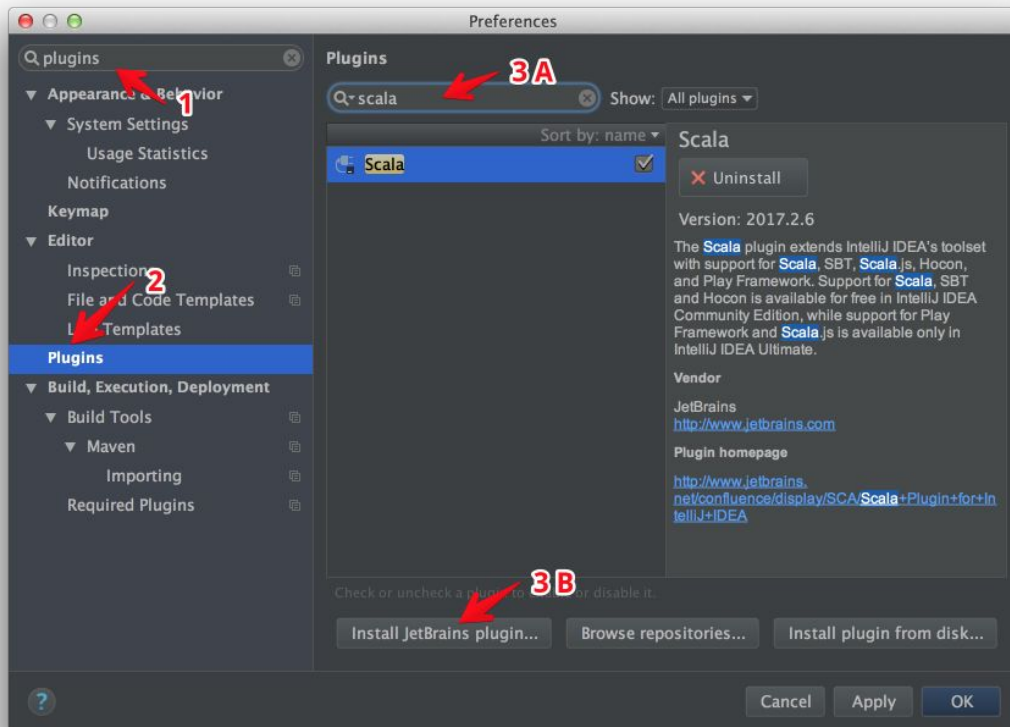
Полезные разделы:

- Keymap
- Color scheme
- Code style
- Plugins
- Scala
- SBT

# Среда разработки и тестирования



## Настройка, установка плагинов



<https://plugins.jetbrains.com/top-downloads/idea>

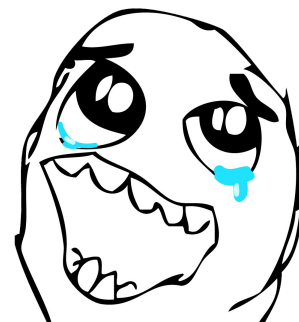


Создание проекта:

- File => New => Project
- File => Open

Навигация:

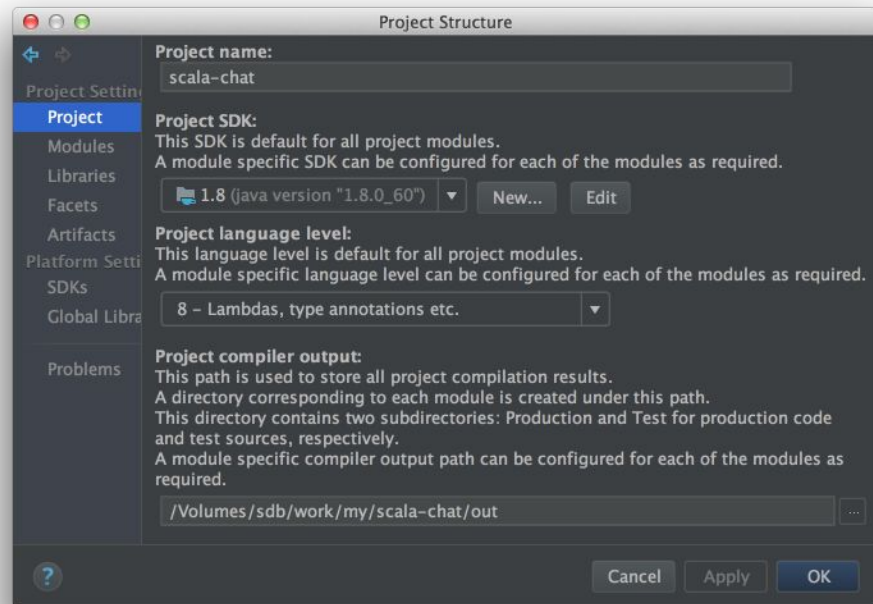
- Ctrl+N, Ctrl+Shift+N, Ctrl+Alt+Shift+N (на маке Ctrl -> Cmd)
- CamelCase-поиск, пробел в конце
- Shift+Shift - поиск везде
- Ctrl+E (Cmd+E) - последние файлы (табы не нужны?)
- Alt+1 (Cmd+1) - левая панель навигации
- и т.д. (см. [хорошее видео](#))
- [официальный хелп](#)





## Настройки проекта

- Как правило, уже всё настроено силами SBT
- Основное для нас здесь - отладка проблем и нестандартные конфигурации:
  - Выбор других SDK
  - Кастомные пути к файлам
  - Зависимые библиотеки







## Шаблоны кода, рефакторинг

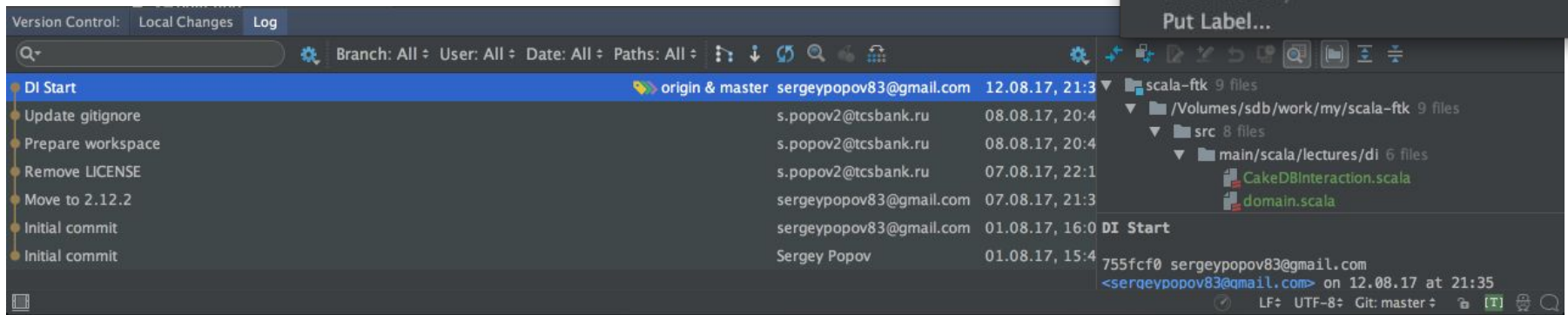
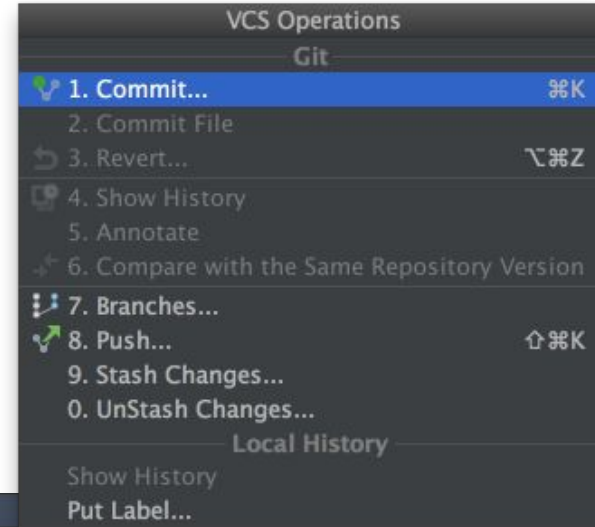
- Live templates: генерация кода по шаблонам
  - <https://www.jetbrains.com/help/idea/live-templates-2.html>
- Refactoring:
  - Наиболее частые примеры:
    - Rename variable, method, class
    - Extract variable, method
  - <https://www.jetbrains.com/help/idea/introduction-to-refactoring.html>
  - <https://www.jetbrains.com/help/idea/refactoring-2.html>

# Среда разработки и тестирования



## Работа с системой контроля версий

- Alt+~ (Ctrl+V), далее цифра - окно действий
- VCS -> Git - полный список действий
- View -> Tool Windows -> Version Control - просмотр текущего состояния, истории



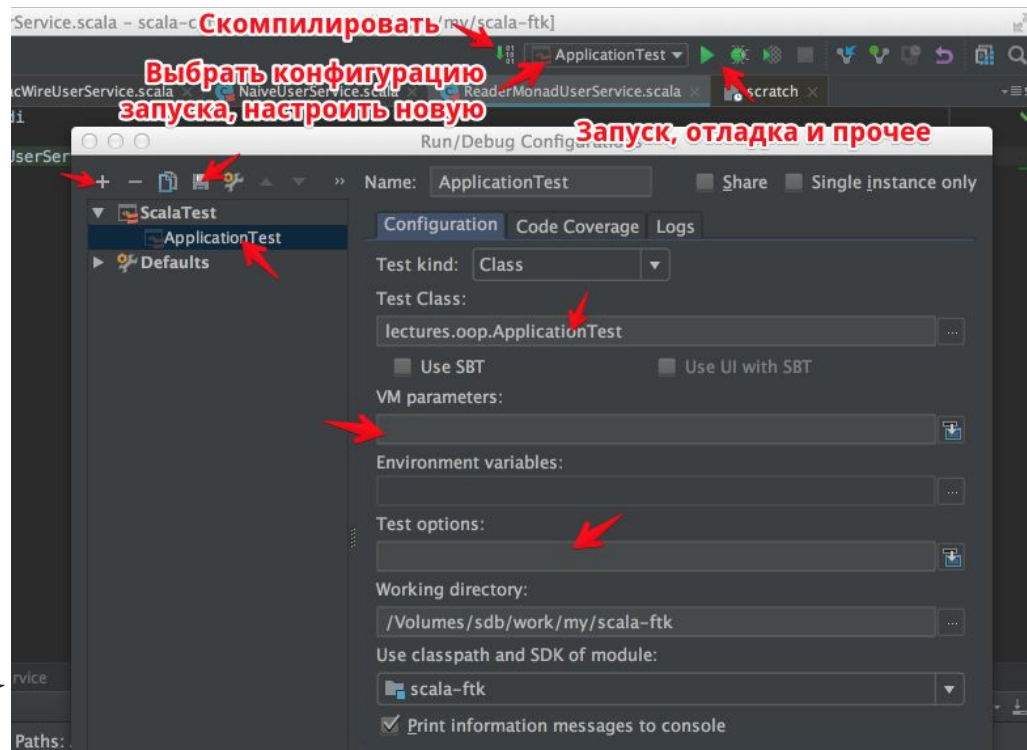
# Среда разработки и тестирования



Запуск, отладка, точки останова  
Тестирование

Полезные кнопки:

- Ctrl+Shift+F10, Ctrl+F10  
(Мак - аналогично) -  
запустить приложение или  
тест под курсором
- F8, F9 - передвижение по  
программе
- Брекпоинты на  
исключения





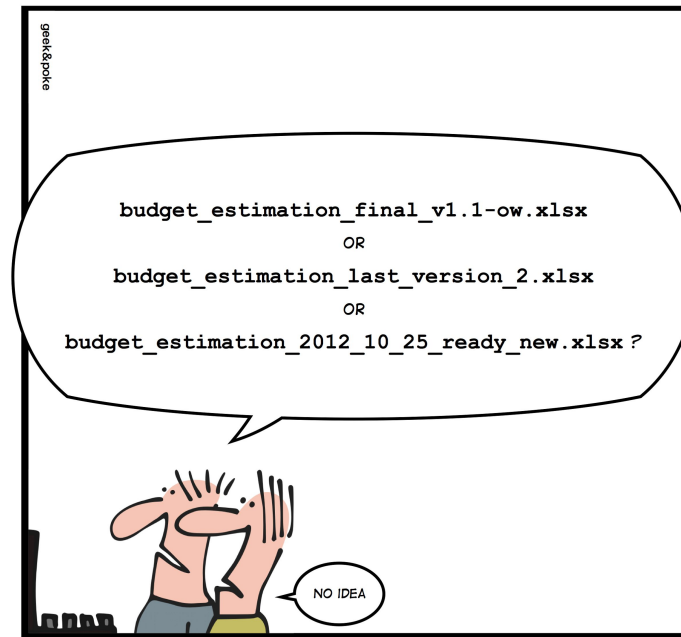
# Системы контроля версий



Зачем оно вообще нужно?

- Частые изменения документа, необходимость возврата к предыдущим версиям
- Параллельная работа над одним и тем же документом
- Совместная работа над документами
  - Актуальные копии последних версий документов
  - Обнаружение и разрешение конфликтов
  - Просмотр истории изменений
- Проверка и контроль над изменениями других
- Автоматизация каких-либо действий в связи с изменениями

## SIMPLY EXPLAINED



VERSION CONTROL



CVS: динозавр-первопроходец. 1986 (shell), 1990 (C) - 2008

- Централизованная
- Версионирование только файлов
- Не отслеживается переименование/перемещение файлов
- Короткоживущие ветки (медленная работа с ветками)
- Нет атомарных коммитов
- Сейчас используют: OpenBSD



Subversion (SVN): фиксим недостатки CVS, 2000 - ...

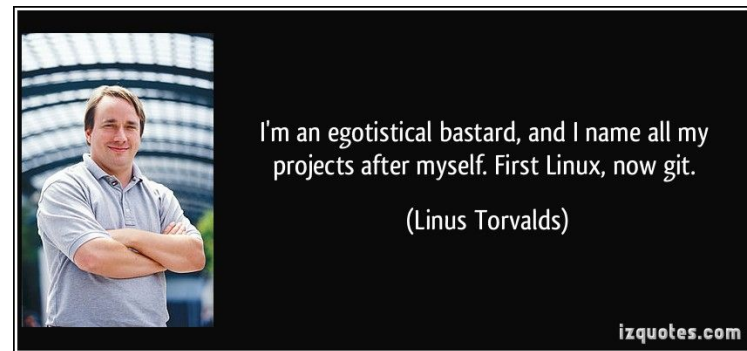
- Совместимость с CVS
- Атомарные коммиты
- Быстрее операции с ветками, но это все еще неудобно
- Тегов нет как таковых. Это всего лишь бранчи.
- Все еще централизованная
- Сравнительно небольшая скорость
- Все еще есть проблемы с переименованием файлов/директорий
- Сейчас используют: clang, gcc, SourceForge, FreeBSD, PuTTY, OpenOffice

# Системы контроля версий. История



## Двигатель прогресса Linux Kernel

- Уже в 1998 году Linus-у было тяжело ревьюить и мержить патчи
- В 2003 году ядро перешло в распределенную систему контроля версий BitKeeper
- Но в 2005 году кончилась халявная лицензия, а вместе с ней плюшки:
  - Распределенный рабочий процесс
  - Автоматизированная работа с патчами
  - История за продолжительный период времени
  - Иерархическая организация разработчиков
  - Скорость работы с большим количеством изменений и файлов
- Matt Mackall и Linus Torvalds запилили Mercurial и Git соответственно
- И началась новая эра священных войн





# Системы контроля версий. История



Общее:

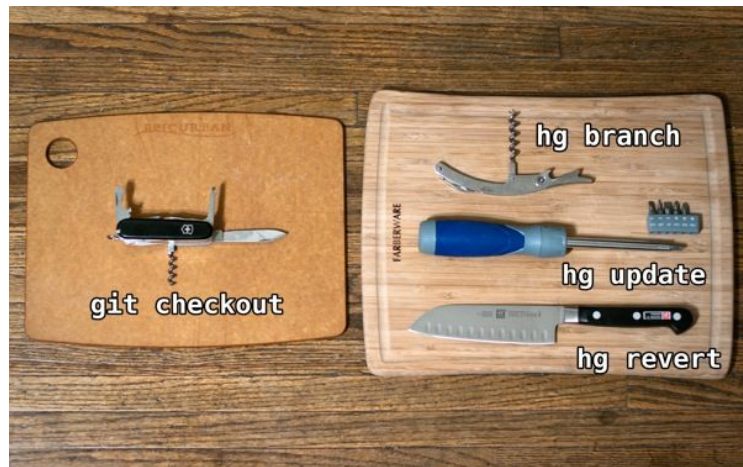
- Ревизии ассоциируются с контрольными суммами
- История - направленный ациклический граф
- Высокоуровневые функции: бисекция, выборочные фиксации

Mercurial:

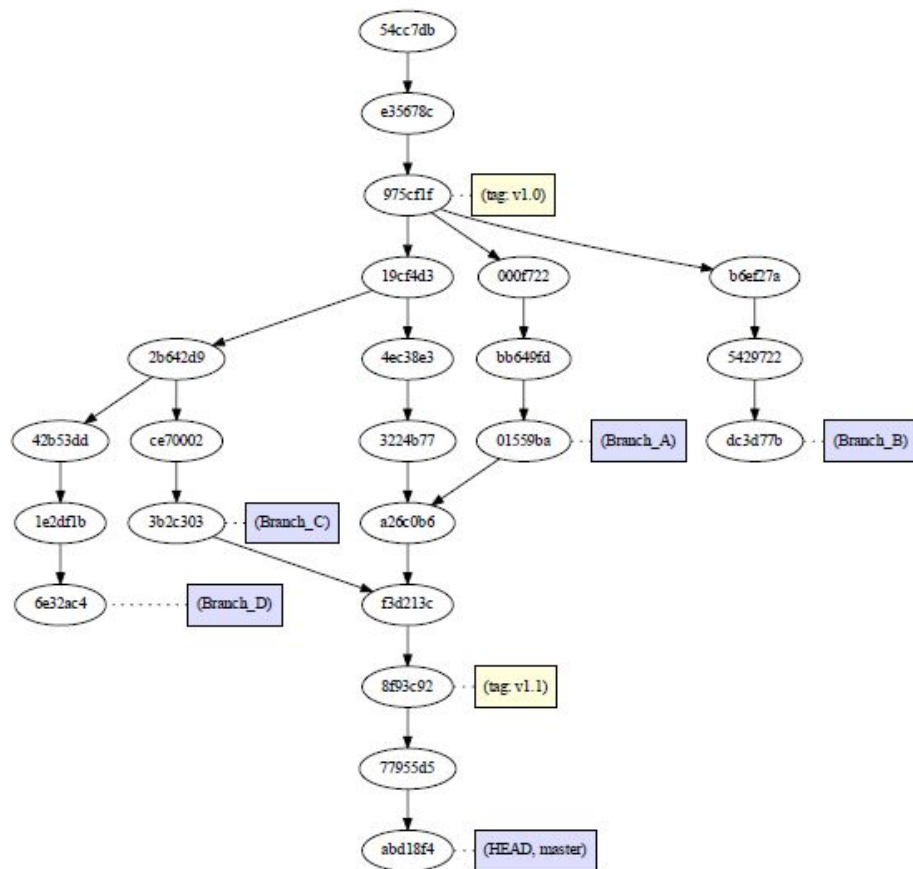
- Revlog: каждый файл имеет индекс и файл с данными, хранятся diff-ы
- Все ветки попадают на сервер и известны всем
- Вся история всегда сохраняется
- Куча несогласованных расширений для решения типичных проблем

Git:

- Каждый файл - это BLOB
- Каждая ревизия - полная копия файлов
- Упаковка данных
- Быстрые коммиты и чекауты
- diff строится на лету
- Ветки не привязываются к коммитам



# Системы контроля версий. Git





## Принципы организации репозитория Git

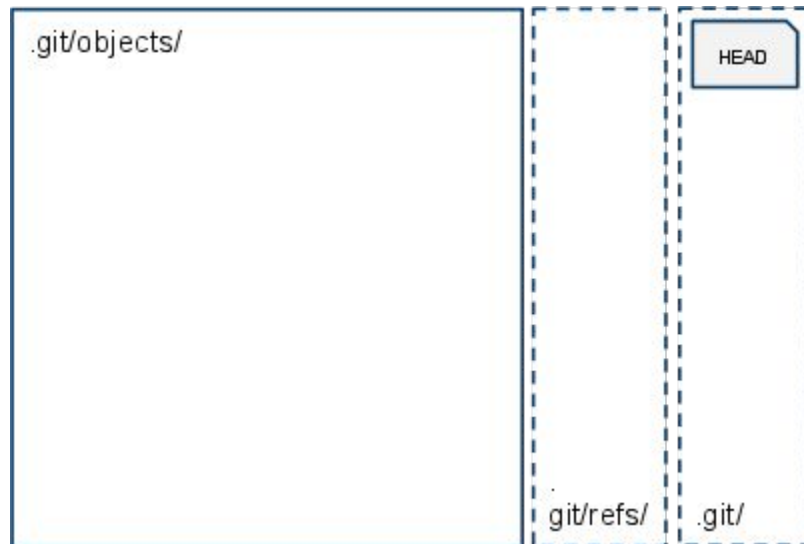
### — Основные сущности:

- Объект (.git/objects/): blob, tree, commit, annotated tag
- Ссылка (.git/refs/): branch, remote branch, lightweight tag, symbolic reference
- Индекс (.git/index)

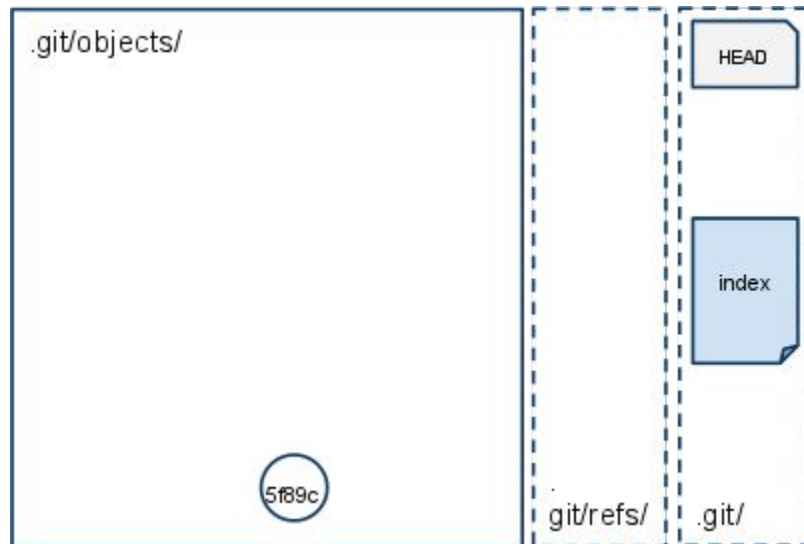
### — Ссылки на почитать:

- <https://habrahabr.ru/post/143079/>
- <http://teohm.com/blog/learning-git-internals-by-example/>

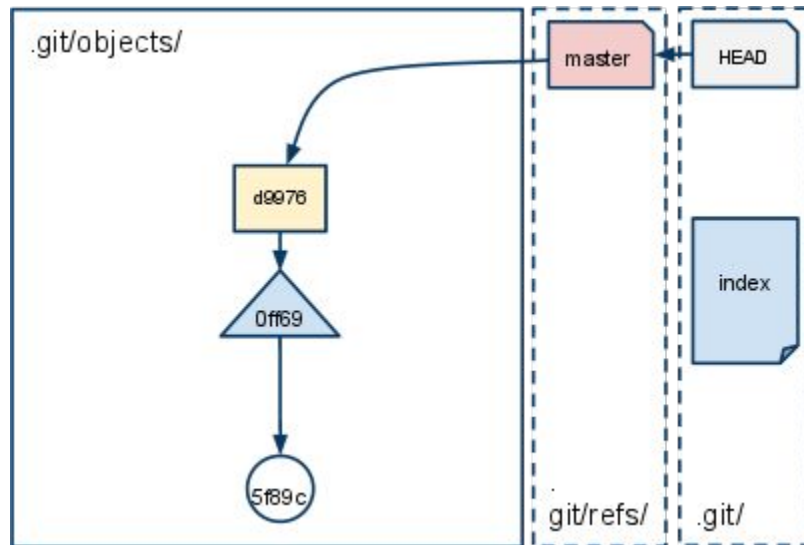
# Системы контроля версий



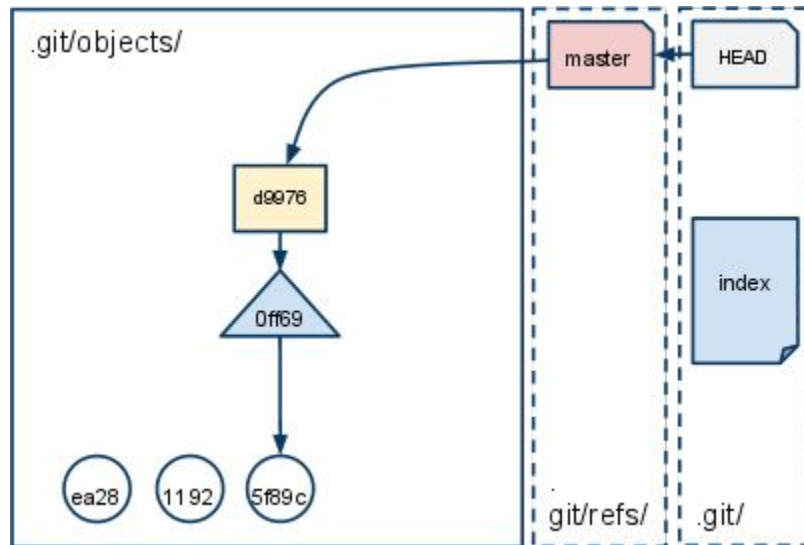
# Системы контроля версий



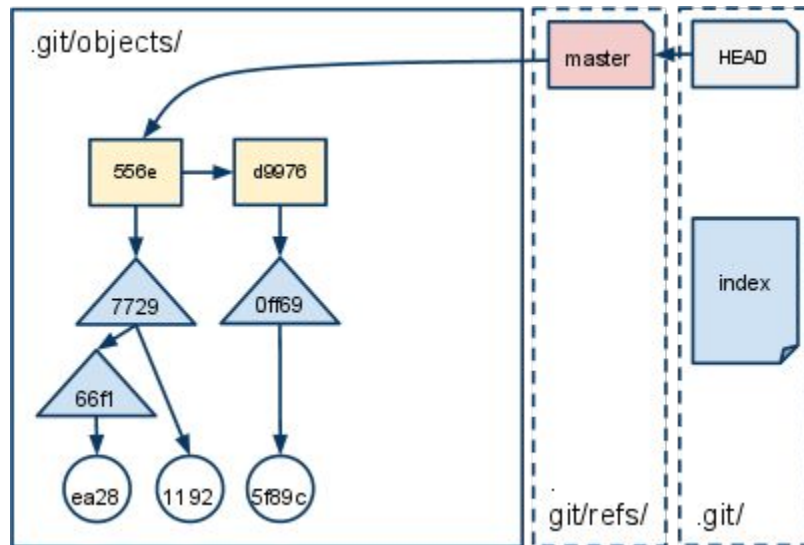
# Системы контроля версий



# Системы контроля версий

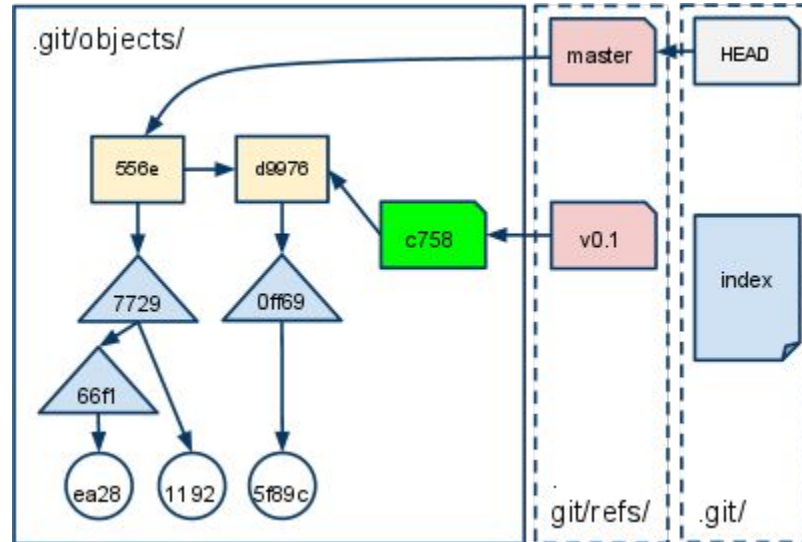


# Системы контроля версий

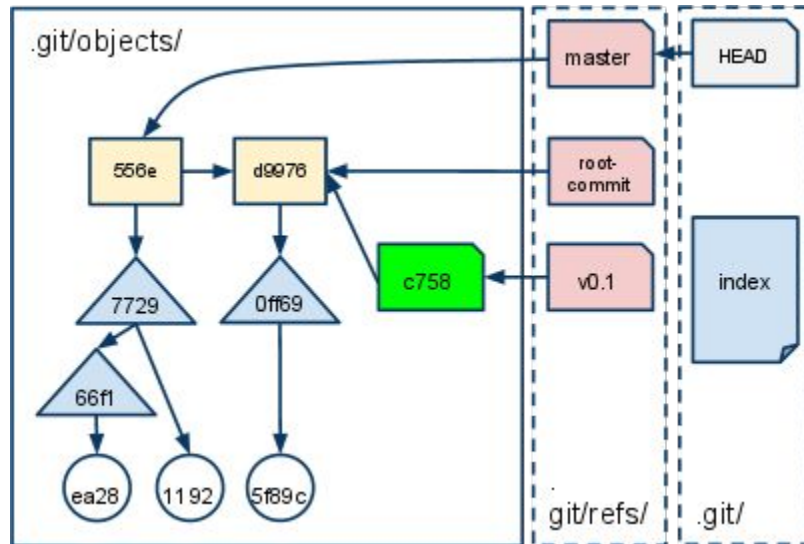




# Системы контроля версий



# Системы контроля версий



# Системы контроля версий. Базовые операции



## Создание нового репозитория

- `git init` - создать пустой репозиторий
- `git clone [url]` - создать из существующего репозитория

## Работа с индексом

- `git add [file]` - добавить файл
- `git reset [file]` - удалить файл

## Сохранение

- `git commit -m [message]` - сохранить изменения из индекса в репозитории

## Откат

- `git revert [commit]` - откатить отдельный коммит
- `git reset [commit]` - откатить на конкретный коммит

## Ветвление

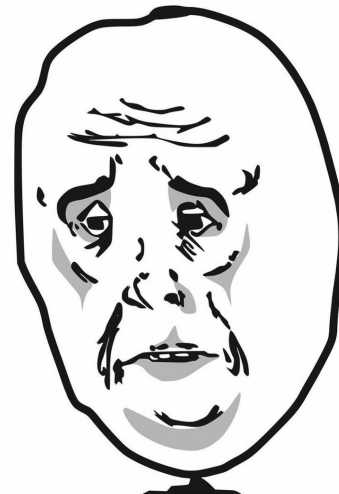
- `git branch` - посмотреть список веток
- `git checkout -b [new_branch]` - создать новую ветку
- `git branch -d [branch]` - удалить ветку

## Объединение

- `git merge [branch]`

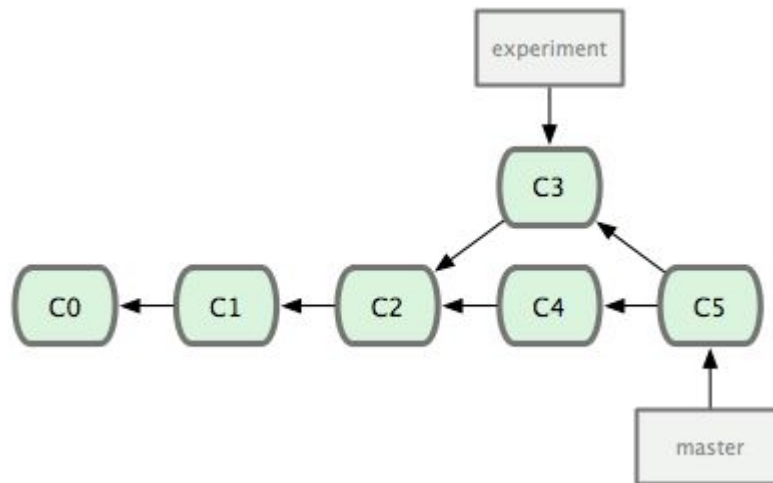
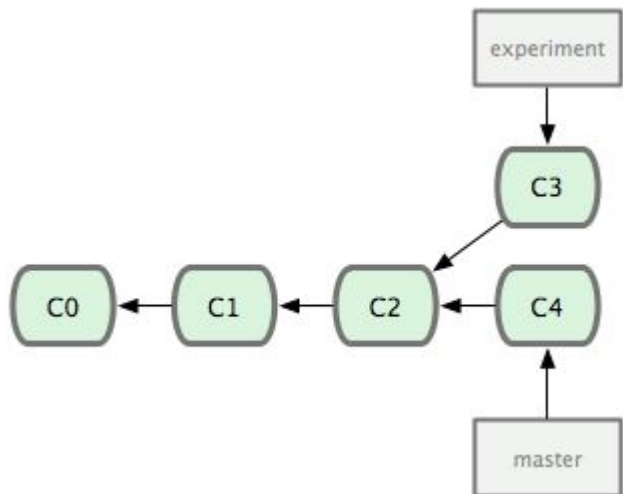
## Тегирование

- `git tag [tag_name]`
- `git tag -m [message] [tag-name]`

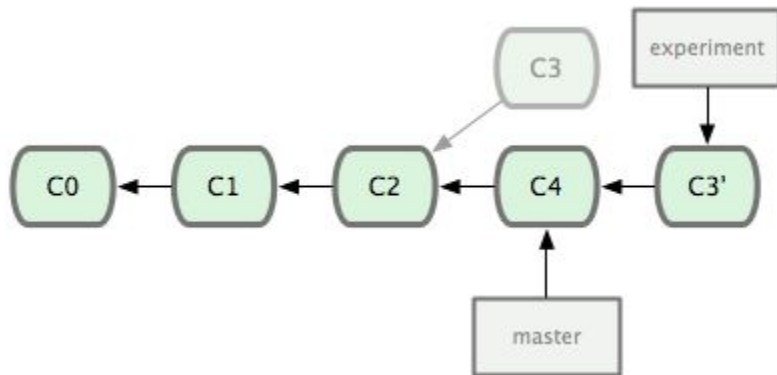


<https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf>

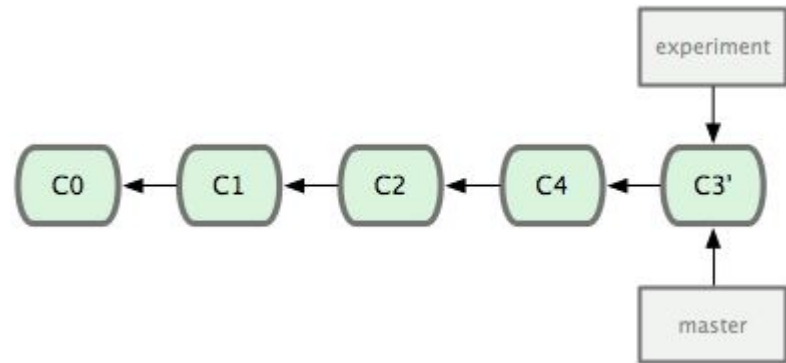
# Системы контроля версий. Rebase



# Системы контроля версий. Rebase



```
$ git checkout experiment  
$ git rebase master
```



```
$ git checkout master  
$ git merge experiment
```



Stash:

- Поможет временно отложить незакоммиченные правки

Rebase:

- rebase onto: см. [git-scm book](#)

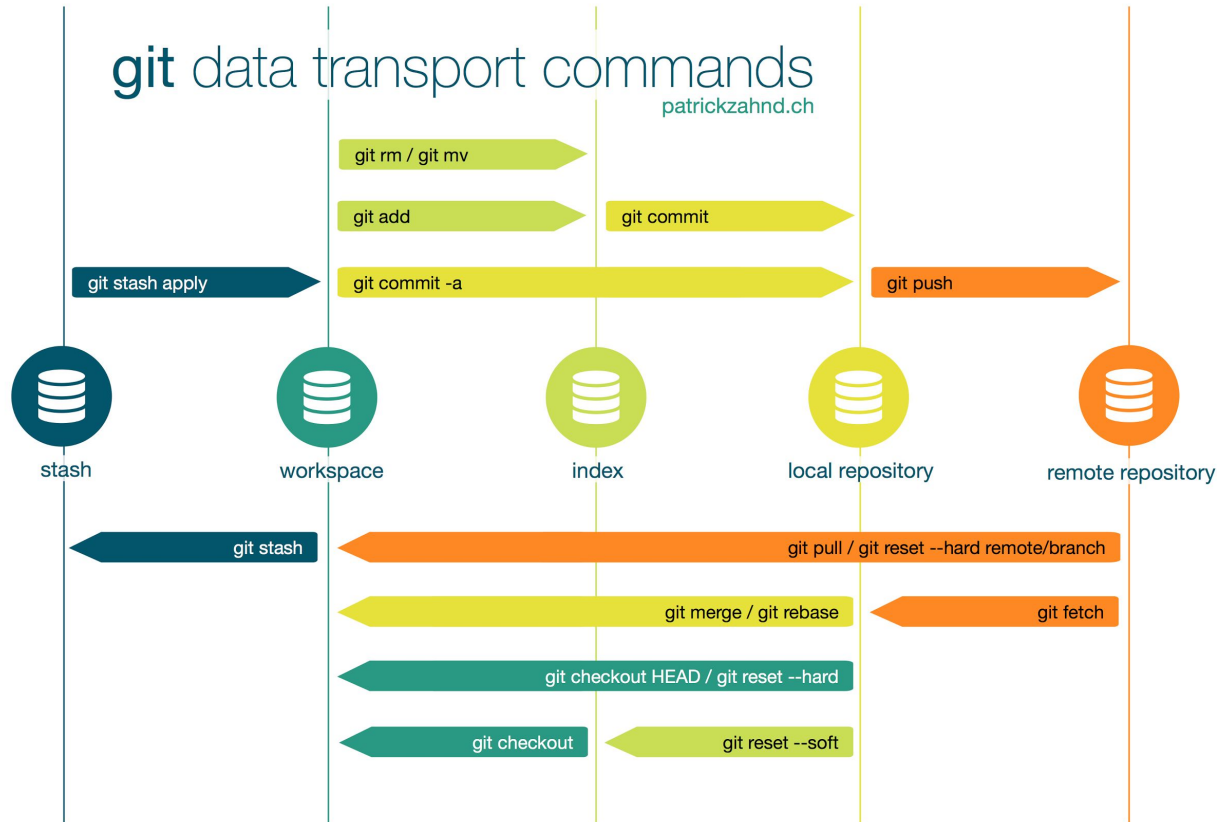
Cherry-pick:

- `git cherry-pick [commit]` - как rebase, только совсем чуть-чуть

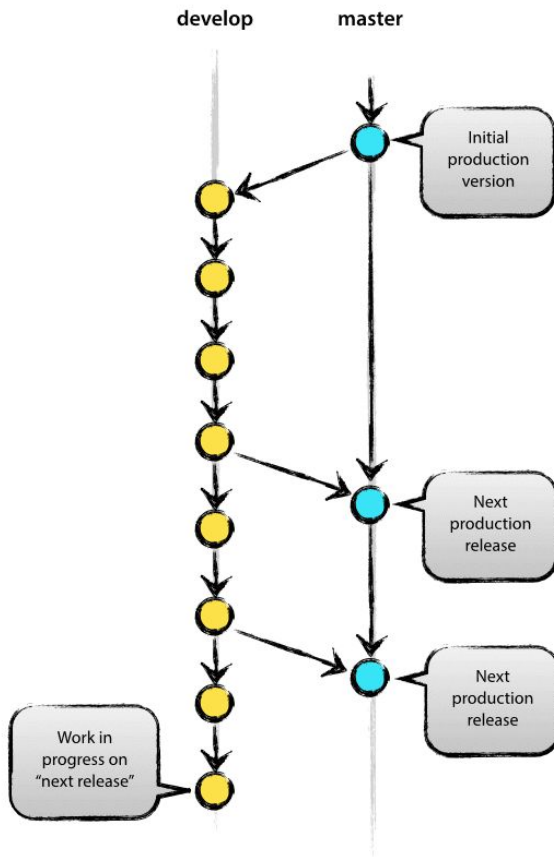
Bisect:

- Хороший способ найти ломающий коммит
- [git-scm book](#)

# Системы контроля версий



# Системы контроля версий. Gitflow Workflow

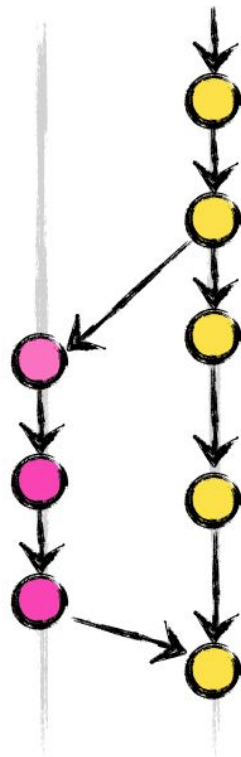




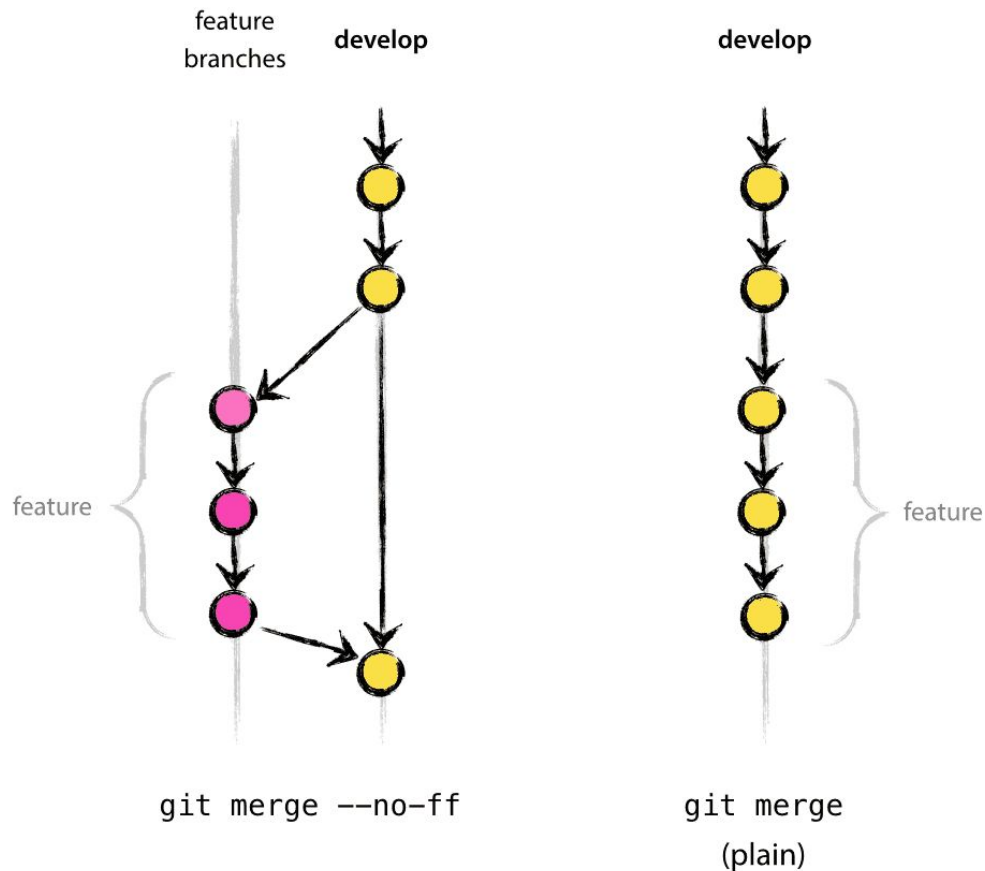
# Системы контроля версий. Gitflow Workflow



feature  
branches      develop



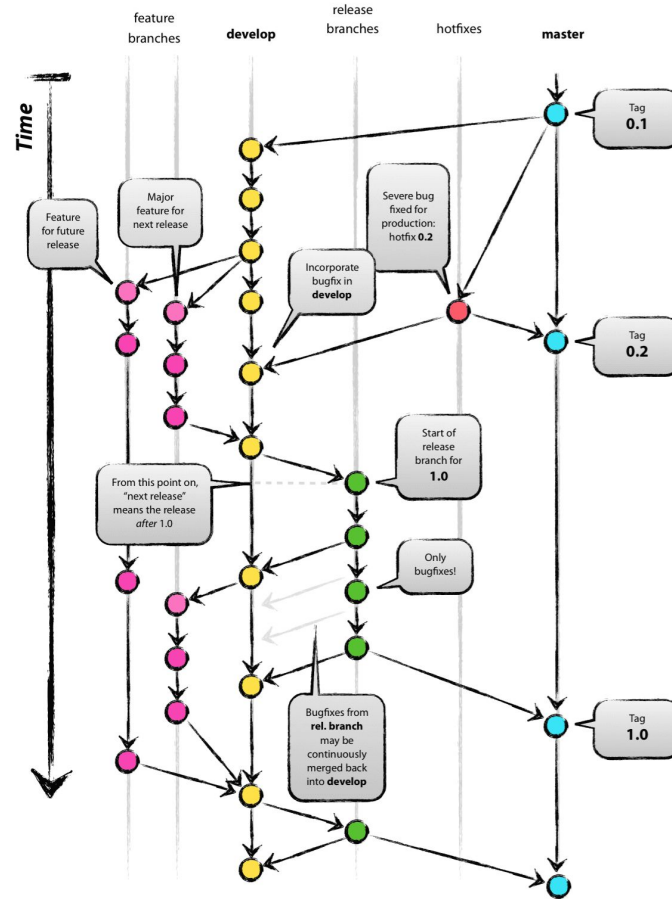
# Системы контроля версий. Gitflow Workflow



# Системы контроля версий. Gitflow Workflow



# Системы контроля версий. Gitflow Workflow





# Сборка проектов. SBT



Немного определений:

- Исходный код - код программы на человекопонятном языке программирования (Scala, Java, C++, ASM etc.)
- Бинарный код - инструкции для центрального процессора
- Байт-код - инструкции для виртуальной машины

Жизненный цикл программы на Java:

1. Написание: разработчик генерирует тонну текстов (\*.scala, \*.java etc.)
2. Компиляция: бравый компилятор преобразует исходный код в байт-код (\*.class)
3. Сборка: сборщик из class-файлов собирает jar-пакеты
4. Исполнение: JVM запускает набор jar-пакетов на исполнение в виртуальной машине



## Сборка java-программы “на коленке”:

```
$ cat src/ru/tinkoff/HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

$ javac HelloWorld.java

$ ls
HelloWorld.class    HelloWorld.java

$ java -classpath . HelloWorld
Hello World!
```



## Сборка java-программы с пакетами:

```
$ vi HelloWorld.java # add "package ru.tinkoff.helloworld;"
$ mkdir -p src/ru/tinkoff/helloworld
$ mkdir bin
$ mv HelloWorld.java src/ru/tinkoff/helloworld
$ javac -d bin src/ru/tinkoff/helloworld/HelloWorld.java

$ find . -type f
./bin/ru/tinkoff/helloworld/HelloWorld.class
./src/ru/tinkoff/helloworld/HelloWorld.java

$ java -classpath ./bin ru.tinkoff.helloworld.HelloWorld
Hello World!
```





## А если надо отдать программу соседу Ваське?

```
$ jar cvf helloworld.jar -C bin .  
added manifest  
adding: ru/(in = 0) (out= 0)(stored 0%)  
adding: ru/tinkoff/(in = 0) (out= 0)(stored 0%)  
adding: ru/tinkoff/helloworld/(in = 0) (out= 0)(stored 0%)  
adding: ru/tinkoff/helloworld/HelloWorld.class(in = 448) (out= 302)(deflated 32%)  
  
$ java -cp helloworld.jar ru.tinkoff.helloworld.HelloWorld  
Hello World!
```



А если у Васи уже много таких модулей?

```
$ mkdir lib

$ mv helloworld.jar lib/

$ java -cp 'lib/*' ru.tinkoff.helloworld.HelloWorld
Hello World!
```

# Сборка проектов. SBT. Автоматизируем



Вначале был make (1976).

Потом был GNU make (1988):

```
JFLAGS = -g
JC = javac
JVM= java
FILE=
.SUFFIXES: .java .class
.java.class:
    $(JC) $(JFLAGS) $*.java
CLASSES = Main.java Class1.java Class2.java Class3.java Class4.java

MAIN = Main

default: classes

classes: $(CLASSES:.java=.class)

run: classes
    $(JVM) $(MAIN)

clean:
    $(RM) *.class
```

## GNU Make

A Program for Directed Compilation



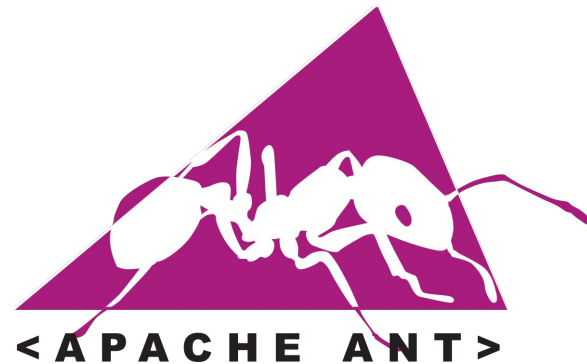
for GNU Make Version 3.79  
by Richard M. Stallman and Roland McGrath

Потом был Ant (2000):

```
<project name="MyProject" default="dist" basedir=". ">
...
  <target name="compile" depends="init"
    description="compile the source">
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
    description="generate the distribution">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
    description="clean up">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

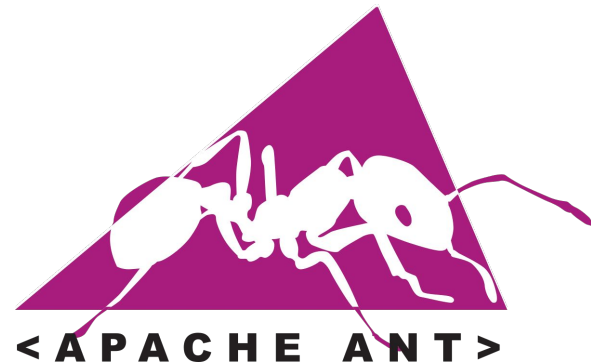


- You just have to write a small file!
- Like Makefile?
- Yes!
- Ok!
- But...

Потом был Ant (2000):

Плюсы:

- Множество готовых функций (task)
- Функции группируются в target
- Можно писать собственные таски
- target зависит от других target-ов



Минусы:

- Огромные XML файлы
- Нет стандартного подхода
- Ручное создание и удаление директорий
- Ручное управление зависимостями



Потом был Maven 2 (2005):

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```



# Сборка проектов. SBT. Автоматизируем



Потом был Maven 2 (2005):



Плюсы:

- Управление зависимостями
  - Репозитории: ivy/ftp/sftp/etc.
- Convention over configuration
- Единые правила генерации и распространения артефактов
- Более простой билд-файл

Минусы:

- Шаг в сторону и надо искать/писать собственный плагин
  - Уже на среднем проекте возникают кучи собственных плагинов

Потом появился Gradle (2007):

```
apply plugin: 'java'
apply plugin: 'checkstyle'
apply plugin: 'findbugs'
apply plugin: 'pmd'

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    testCompile group: 'org.hamcrest', name: 'hamcrest-all', version: '1.3'
}
```







Потом появился Gradle (2007):



Плюсы:

- Собственный DSL на Groovy
- Convention over configuration
- Поддержка Java, Scala, C/C++, Python, etc.

Минусы:

- Собственный DSL на Groovy



И, наконец, пришел SBT (2011):

```
# in build.sbt

lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalaVersion := "2.12.3",
    libraryDependencies ++= Seq(
      "org.scalatest" % "scalatest_2.11" % "2.2.4" %
"test",
      "net.databinder.dispatch" %% "dispatch-core" %
"0.11.3"
    )
  )

# in project/build.properties

sbt.version=1.0.1
```

# Сборка проектов. SBT. Автоматизируем



И, наконец, пришел SBT (2011):

Плюсы:

- Собственный DSL на Scala
- Convention over configuration
- Интерактивная консоль
- Инкрементальная компиляция

Минусы:





Последние версии SBT:

- 0.13.x (2014+)
  - Scala 2.10
  - Два формата проектов (старый и новый)
- 1.0.x (2017):
  - Scala 2.12, JDK 8
  - Убрали много deprecation-ов
  - Новый инкрементальный компилятор
  - Параллельная загрузка артефактов



Как ставить SBT - [здесь](#)

Создание нового проекта из шаблона Giter8:

```
$ sbt new sbt/scala-seed.g8
```

Список шаблонов можно посмотреть [здесь](#).

Запуск проекта:

```
$ sbt run
```

Интерактивная консоль (Scala REPL):

```
$ sbt console
```

Другие полезные команды - [здесь](#).



## Структура проекта

```
build.sbt      # Main SBT build definition file
project/
  build.properties  # Properties for SBT
  Dependencies.scala # Helper objects for the build definition
src/
  main/
    resources/ # Files to include in main jar here
    scala/     # Main Scala sources
    java/      # Main Java sources
  test/
    resources # Files to include in test jar here
    scala/    # Test Scala sources
    java/     # Test Java sources
target/      # Here come files generated during build
```



## Конфигурация билда:

```
lazy val root = (project in file(".")).
  settings(
    List(
      organization := "ru.tinkoff.helloworld",
      scalaVersion := "2.12.1",
      version      := "0.1.0-SNAPSHOT"
    ),
    name := "HelloWorld",
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.1" % Test
  )
```

Key - ключ, хранитель значения, может быть одним из:

- SettingKey[T] - параметр (вычисляется однажды, при старте)
- TaskKey[T] - задача (вычисляется по мере необходимости, скорее всего имеет побочные эффекты)
- InputKey[T] - задача с пользовательским вводом

## Объявление собственных задач:

```
val personName = settingKey[String]("Name of person to geet")
val greet = taskKey[String]("Greet person")

greet := {
  println(s"Hello, ${personName.value}")
  s"I have said hello to ${personName.value}"
}

lazy val root = (project in file(".")).
  settings(
    name := "HelloWorld",
    personName := "Alice"
  )
```

## Запускаем:

```
[sbt]> greet
Hello, Alice
```



# Сборка проектов. SBT



Области определения (scopes): каждый ключ (setting или task) имеет собственную область определения, в рамках которой можно его прочитать/запустить.

Области определения имеют 3 независимые оси (scope axes):

1. project (subproject) - проект
2. dependency configuration - конфигурации сборки: compile, test, runtime.  
Конфигурации могут расширять другие конфигурации, например:  
test -> runtime -> compile
3. task - отдельная задача

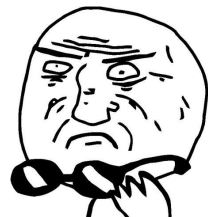
Полный префикс ключа:

```
{<build-uri>}<project-id>/config:task::key
```

Дефолтные префиксы ключа:

```
{ . }/*:key или */*:key
```

Еще примеры задания областей определения можно посмотреть [здесь](#).



Примеры и задачи на различные области определения - [здесь](#).

```
val greet = taskKey[Unit]("Greet you via console")

greet := println("Just greet")
greet in Global := println("Greet in Global")
greet in ThisBuild := println("Greet in ThisBuild")

lazy val root = (project in file("."))
  .settings(
    name := "HelloWorld",
    greet := println("Greet in project"),
    greet in Test := println("Greet in Test"),
    greet in (ThisBuild, Test) := println("Greet in ThisBuild Test"),
    greet in (ThisBuild, Test, compile) :=
      println("Greet in ThisBuild Test compile")
  )
```

// greet  
// \*/\*:greet  
// {./}/\*:greet  
  
// greet  
// test:greet  
// {./}/test:greet  
// {./}/test:compile::greet



Пример с добавлением области определения конфигурации:

```
def isSerial(name: String) = name.endsWith("ISpec")

lazy val Serial = config("serial") extend Test
lazy val serialSettings = inConfig(Serial)(
  Seq(
    fork := true,
    testOptions := Seq(Tests.Filter(isSerial)),
    parallelExecution := false
  ) ++ Defaults.testTasks
)

lazy val root = (project in file("."))
  .configs(Serial)
  .settings(
    name := "HelloWorld",
    serialSettings,
    testOptions in Test := Seq(Tests.Filter(!isSerial(_))),
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.1" % Test
  )
```



## Неуправляемые зависимости

- лежат в `unmanagedBase`, по умолчанию это `lib`

- можно переопределить:

```
unmanagedBase := baseDirectory.value / "custom_lib"
```

- посмотреть все `jar`-ники в неуправляемых зависимостях:

```
unmanagedJars
```

```
[IJ]> show unmanagedBase  
[info] /Volumes/sdb/work/helloworld/lib
```

```
[IJ]> show unmanagedJars  
[info] * Attributed(/Volumes/sdb/work/helloworld/lib/scalatest_2.12-3.0.4.jar)
```



Управляемые зависимости:

- это зависимости, которые автоматически подгружаются с внешних серверов при сборке проекта
- используется maven-like репозитории
- основной ключ, хранящий управляемые зависимости - `libraryDependencies`
- возможна автоматическая подстановка версии Scala оператором `%%`

```
libraryDependencies += groupId % artifactID % revision % configuration
libraryDependencies += Seq(
  "org.xerial" % "sqlite-jdbc" % "3.7.2",
  "org.scalatest" % "scalatest_2.12.2" % "3.0.1" % "test",
  "org.scalatest" %% "scalatest" % "3.0.1" % "test"
)
```



Версии артефактов можно указывать неточно ([примеры настроек](#)):

```
libraryDependencies += Seq(  
  "org.scalatest" %% "scalatest" % "3.0.1" % "test",  
  "org.scalatest" %% "scalatest" % "3.0.+" % "test",  
  "org.scalatest" %% "scalatest" % "[3.0,)" % "test"  
)
```



Резолверы позволяют SBT найти артефакты не только в стандартном maven-репозитории, но и в любых других. Задаются они так:

```
resolvers += "Sonatype Snapshots" at "https://oss.sonatype.org/content/repositories/snapshots"
```

Параметр `externalResolvers` содержит в себе все резолверы из `resolvers` плюс резолверы по умолчанию. Если надо убрать резолверы по умолчанию, то можно переопределить параметр `externalResolvers`.



Билды с несколькими проектами:

```
lazy val commonSettings = Seq(  
  organization := "ru.tinkoff",  
  version := "0.1.0-SNAPSHOT",  
  scalaVersion := "2.12.3"  
)  
  
lazy val helloworld = (project in file("helloworld"))  
  .settings(  
    commonSettings,  
    // other settings  
  )  
  
lazy val util = (project in file("util"))  
  .settings(  
    commonSettings,  
    // other settings  
  )
```



## Зависимости между проектами. Простое объединение:

```
// run all tasks on all projects
lazy val root = (project in file("."))
  .aggregate(helloworld, util)
  .settings(
    aggregate in update := false // except for update tasks
  )
```

## Если код одного проекта зависит от кода другого проекта:

```
// General classpath dependencies
lazy val helloworld = project.dependsOn(util)
lazy val helloworld = project.dependsOn(core, util)

// Per-configuration dependencies
lazy val helloworld = project.dependsOn(util % "test") // means test->compile
lazy val helloworld = project.dependsOn(util % "test->test")
lazy val helloworld = project.dependsOn(util % "test->test;compile->compile")
```



Плагины позволяют легко добавлять новые настройки и задачи к билду.

Например, для добавления расчета покрытия кода достаточно:

```
// 1. Include plugin in project/plugins.sbt:  
addSbtPlugin("org.scoverage" % "sbt-scoverage" % "1.5.1")  
  
// 2. Execute in command line  
$ sbt clean coverage test coverageReport  
...  
[info] Written HTML coverage report  
[/Volumes/sdb/work/scala-ftk/target/scala-2.12/scoverage-report/index.html]  
[info] Statement coverage.: 17.19%  
[info] Branch coverage....: 0.00%  
[info] Coverage reports completed  
[info] All done. Coverage was [17.19%]
```

В отдельных случаях при добавлении плагина надо еще добавить нужный резолвер.



Плагины бывают автоконфигурируемые или с ручной конфигурацией. В последнем случае необходимо включить плагин и/или изменить некоторые значения:

```
lazy val helloworld = (project in file("helloworld"))  
  .enablePlugins(FooPlugin, BarPlugin)  
  .disablePlugins(plugins.IvyPlugin)  
  .settings(  
    enableBarSuperFeature := true  
  )
```

Посмотреть список подключенных плагинов: `plugins`

Как написать свой плагин - [см. здесь](#)

Большой список плагинов для SBT - [здесь](#)