**Министерство образования и науки РФ**

**ФГБОУ ВО «АНГАРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**Факультет технической кибернетики**

**Кафедра вычислительные машины и комплексы**

**КУРСОВАЯ РАБОТА**

**По дисциплине:** «Современные технологии проектирования программных комплексов»

**Тема:** «Реализация подхода Scrum в GitHub: Распределённый GitHub в версиях консольных приложений»

**Выполнил:**

Студент группы ИВТм-17-1

Борискин Б.Б.

**Проверил:**

к.т.н., доцент Кривов М.В.

Ангарск, 2017 год

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 3](#_Toc504420842)

[Глава 1 Основы современных технологий проектирования в разрезе метода scrum agile и хостинг репозитория GitHub. 4](#_Toc504420843)

[**1.** **Agile, манифест разработки программного обеспечения** 4](#_Toc504420844)

[**2.** **Методология гибкой разработки Scrum** 5](#_Toc504420845)

[**3.** **Репозиторий Git, система управления версиями** 8](#_Toc504420846)

[Глава 2 Создание публичного репозитория GitHub, принципы и требования 14](#_Toc504420847)

[**1.** **Public GitHub в хостинг приложении, создание, протоколы** 14](#_Toc504420848)

[**2.** **GitHub Desktop, Git Bash: Основы работы** 17](#_Toc504420849)

[**3.** **Распределённый GitHub в консольных версиях C# Visual Studio, Scrum методология** 20](#_Toc504420850)

[Выводы 29](#_Toc504420851)

# **ВВЕДЕНИЕ**

Современный этап проектирования программных комплексов и систем позиционирует себя, прежде всего, со стремлением наиболее плотного и эффективного сотрудничества непосредственно с пользователями (заказчиками) программного продукта. Но этого становиться недостаточно, сложность любого проекта различных отраслей деятельности человека порождает необходимость совместной работы в том числе и специалистов, имеющих непосредственное отношение к проектированию и созданию программного кода. Совместная работа авторов крупных проектов и программистов определяет и позволяет достигнуть:

* высокого уровня технологической поддержки процессов разработки и сопровождения программного обеспечения;
* положительного воздействия на такие факторы, как: производительность, качество продукции, соблюдение стандартов, документирование;
* относительно приемлемый уровень экономического эффекта инвестиций в проект.

Рассматривая различные методы и подходы к проектированию программных комплексов, стоит отметить разнообразие и сложность систем, именуемых в международной научно-технической практике «Систем, интенсивно использующих программное обеспечение» (Software Intensive System). […]. Функциональный потенциал последних определяется программным обеспечением, либо имеет ряд зависимостей от него, либо взаимодействует с другими компонентами и подсистемами программ, датчиками, людьми и приборами. Всё это приводит к определённым проблемам в ходе работы над сложными программными комплексами:

* отсутствие метрик и данных по проектам и процессам разработки программного комплекса;
* снижение продуктивности и полное отсутствие отдачи в следствии длительность разработки;
* не соответствие процессов и методов разработки программного комплекса;

Выполняя заданные алгоритмы автоматической обработки данных, понять, какие функции выполняет та или иная программа позволяет программная документация. Однако модифицируя программный код для оптимизации можно упустить важные моменты и составляющие, что потребует возврата к исходной точки изменений. Современные технологии и особенности разработки программных средств имеют ряд особенностей, неся в себе неформальный характер требований к исполняемым задачам и творческий подход. Это, в свою очередь, привело к такому подходу как RAD (Rapid Application Development):

* небольшая группа разработчиков, выполнение работ по проектированию отдельных модулей программного обеспечения;
* краткосрочный производственный график (не более трёх месяцев);
* цикличность запросов и реализация последних в результате взаимодействия к конечным пользователем;

Однако данный подход, имея в жизненном цикле программного обеспечения четыре стадии может нивелировать все усилия на стадии реализации и внедрения, так как проектирование потребует значительного временного диапазона, а требования могут измениться. Хотя использование инструментальных CASE (Computer Aided Software Engineering) средств позволяет избежать данных проблем, путём задания единого графического языка, построением диаграмм, генерацией исходного кода для различных платформ, созданием репозитория. Изучение и использование всех современных методов и подходов в реализации качественного программного продукта позволит команде разработчиков наиболее быстро и эффективно провести внедрение программного комплекса избежав при этом досадных ошибок.

Цель данной курсовой работы, рассмотреть особенности современных методов и подходов к разработке программных комплексов, основанных на системе Agile, методах гибкой разработки Scrum, использовании репозитория GitHub – позволяющего реализовать творческий и коллективный подход к реализации проектов, и помимо этого управлять версиями программного продукта.

# **Глава 1 Основы современных технологий проектирования в разрезе метода scrum agile и хостинг репозитория GitHub.**

1. **Agile, манифест разработки программного обеспечения**

Гибкость метода создания программной системы, по принципам Agile заключается прежде всего во взаимодействии между людьми: сотрудниками, клиентами. Представляя собой несколько жёстких финишных этапов работы – спринтов, метод позволяет команде постоянно оценивать результаты выполненной работы и получать отзывы от заказчиков и других участников проекта. Данный подход даёт возможность быстро вносить изменения в проект.

История эволюционного управления проектами и адаптивная разработка программного обеспечения берёт своё начало в 1970-х годах. Доктор Уинстон Ройс (Winston Walker Royce, 1929-1995 гг.)[[1]](#footnote-1) представляет документ под названием «Управление развитием крупных программных систем». Критика заключалась в том, что программное обеспечение не должно разрабатываться последовательно, начать новую фазу без окончания предыдущей невозможно. Доктор Ройс предлагал использовать фазовый подход, в котором разработчики в первую очередь собирают все необходимые требования проекта, затем завершают свою архитектуру и дизайн, а затем записывают весь код.

В 1990-х годах уже существовали некоторые гибкие методы разработки программного обеспечения. К ним можно отнести:

* RAD (Rapid Application Development) – быстрая разработка приложений, 1991 год;
* DSDM (Dynamic System Development Method) – метод разработки динамических систем, 1994 год;
* Методология SCRUM – реализация Agile, разделение работ на короткие итерации[[2]](#footnote-2), 1995 год;
* Crystal Clear и экстремальное программирование (XP) – 1996 год;
* FDD (Feature driven development) – разработка, управляемая функциональностью, 1997 год;

Февраль 2001 года значим событием, произошедшим на курорте Snowbird в штате Юта, состоялось собрание семнадцати разработчиков для обсуждения лёгких метод разработки. Был опубликован Манифест о гибкой разработке программного обеспечения Agile. Манифест содержит четыре основополагающих идеи:

1. Люди и взаимодействие важнее процессов и инструментов.
2. Рабочее программное обеспечение важнее документации.
3. Сотрудничество с клиентами важнее согласования условий контракта.
4. Готовность внести изменения в приоритете, нежели придерживаться первоначального плана.

А также двенадцать принципов, предполагающих максимально эффективно управлять проектами:

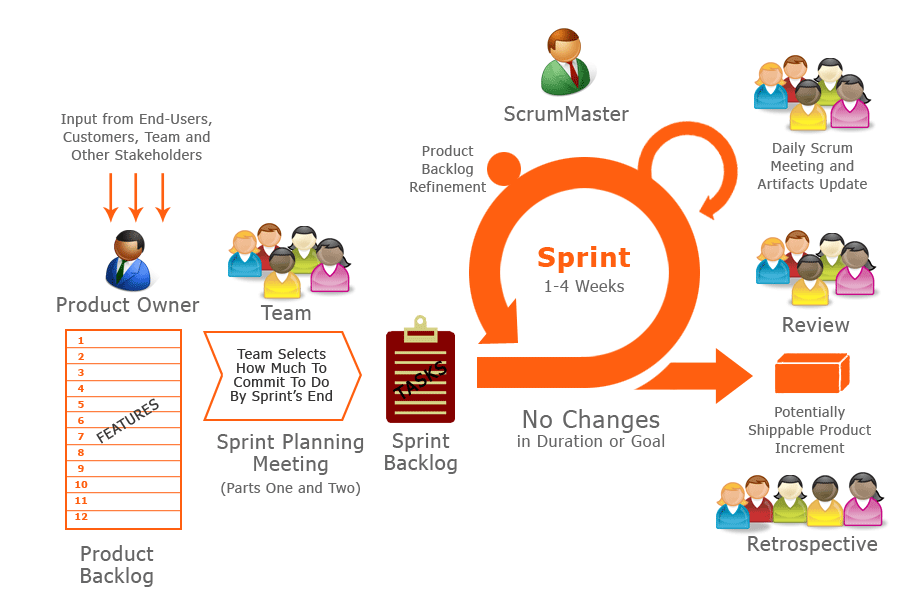
1. Удовлетворённость клиентов за счёт ранней и непрерывной поставки программного обеспечения. Клиенты более счастливы, когда они получают рабочее программное обеспечение через регулярные промежутки времени.
2. Вносить изменения требований к продукту на протяжении всего процесса разработки.
3. Частая поставка рабочего программного обеспечения (каждый месяц, две недели, неделю и т.д.).
4. Сотрудничество между заинтересованными сторонами (заказчиком и разработчиком) на протяжении всего проекта.
5. Поддержка, доверие и мотивация вовлечённых людей. Мотивированные команды с большей вероятностью выполняют свою лучшую работу, чем сотрудники, недовольные условиями труда.
6. Взаимодействие лицом к лицу. Коммуникация более успешна, когда команды разработчиков имеют возможность общаться напрямую.
7. Рабочее программное обеспечение является основной мерой прогресса. Представление функционального программного обеспечения клиенту является конечным фактором, который измеряет прогресс.
8. Поддержка постоянного темпа работы. Команды устанавливают повторяемую и поддерживаемую скорость работы, с которой они могут доставлять функционирующее программное обеспечение.
9. Внимание к техническим деталям и дизайну. Правильные навыки и хороший дизайн позволяют команде поддерживать темп, постоянно совершенствовать продукт и работать над изменениями.
10. Простота.
11. Самоорганизующиеся команды поощряют отличную архитектуру, требования и проекты. Квалифицированные и мотивированные члены команды, которые обладают полномочиями принимать решения, регулярно общаются с другими членами команды и обмениваются идеями, которые обеспечат создание качественного продукта.
12. Постоянная адаптация к изменяющимся условиям, что поможет сделать продукт более конкурентоспособным на рынке.

Помимо идей и принципов гибкая методология управления проектами Agile предусматривает ряд ключевых моментов:

1. Визуальный контроль. Участники проекта в ходе работы над проектом используют карточки различных цветов и видов, которые сигнализируют, какой элемент конечного продукта уже разработан, спланирован, завершён и т.д. Таким образом, команда наглядно имеет представление о существующем положении дел. Визуальный контроль обеспечивает одинаковое видение проекта каждым участником.
2. Все участники проекта работают рядом, включая клиента. Такой подход не только ускоряет многие процессы, связанные с информированием участников рабочей группы, но и создаёт благоприятную атмосферу для сотрудничества и эффективной работы.
3. Адаптируемое управление. Руководитель проекта – не человек, который раздаёт указания, а лидер, определяющий основные правила работы и сотрудничества.
4. Совместная работа. Команда, руководитель проекта и клиент работают сообща, что исключает возможность потери информации и непонимания целей. Также прозрачность всех процессов позволяет моментально исключать появившиеся проблемы и находить удачные решения и улучшения.
5. Работа, основанная на разделении общего объёма проекта на составные части. Такая система работы значительно снижает сложность проекта и позволяет командам сфокусироваться на каждой части отдельно.
6. Работа над ошибками. В ходе работы одного цикла команда осваивает новые навыки и анализирует произошедшие ошибки, что исключает их появление в следующем цикле.
7. Спринты и ежедневные встречи. Спринты – отрезки времени, за которые команда выполняет ряд задач.
8. **Методология гибкой разработки Scrum**

Scrum – является одной из гибких методологий разработки Agile. Термин заимствован из спорта. В регби, перед началом игры, участник образуют определённую фигуру. Поводом для становления и развития данной методологии послужили отраслевые требования к стремительно изменяющейся модели бизнес процессов, необходимостью их автоматизации, более глубокого анализа, перевода отношений клиент-сотрудник в плоскость удалённого взаимодействия. В середине 90-х годов, программное обеспечение носило дескотопный[[3]](#footnote-3) характер. Появление веб-приложений оптимизировало функциональность, что усилило конкуренцию между компаниями. Применяя новейшие технологии – выигрываешь рынок и клиентов.

Изображение 1 Процесс Scrum



Владелец продукта, определяющий необходимые функции, ставит команде разработчиков определённые задачи. Команда, в свою очередь, определяет количество необходимое количество спринтов до окончания работ. Однако стоит отметить, что данная методология, согласно схеме, позволяет на любом этапе исправить отставание в разработке продукта без изменения продолжительности разработки, ежедневно проводя встречи, ретроспективный анализ и обновление артефактов:

1. ***Product backlog[[4]](#footnote-4)*** 
   1. Список всех требований, необходимых для выполнения проекта. Отсутствие требований в Backlog’е завершает проект.
   2. Все требования описаны по единому шаблону, который называют User Story (пользовательская история).
   3. Требования составляются таким образом, что очевидно и понятно, какова ценность последних для пользователя.
   4. Сортировка требований по приоритетам, пересматриваемых каждый спринт.
2. ***Sprint backlog[[5]](#footnote-5)***
   1. Список всех требований, которые необходимо сделать в ближайший спринт.
   2. В течение спринта, новые требования не могут появиться в Sprint backlog.
   3. Все требования должны быть разделены на задачи и оценки.

Иначе, Sprint Backlog можно рассматривать как обязательство команды к тому, что они должны сделать и выполнить в ближайшее время. Каждое требование разделяется на задачи, представленные на Kanban-доске.

1. ***Sprint Goal[[6]](#footnote-6)***
   1. Кратко описывается то, ради чего выполняется данный спринт.
   2. Целенаправленность на спринт помогает команде принимать обоснованные решения.

Данные артефакт необходим команде проекта для самостоятельного принятия решения, в случае появления альтернативных путей решения поставленной задачи. Однако Product Owner (владелец продукта) определяя цель спринта, способствует более глубокому пониманию и осознанности решений команды.

1. ***Sprint Burndown Chart[[7]](#footnote-7)***
   1. В качестве «сгорающих» элементов выступают человеко-часы и идеальные единицы (Story Points).
   2. Обновление диаграммы по завершении какой-либо задачи.

Помимо артефактов, Scrum предусматривает исполняющие роли, их не более трёх, однако каждая вносит определённый вклад в разработку проекта:

1. ***Product Owner[[8]](#footnote-8)*** - представитель подразделения, владеющего разрабатываемым продуктом. Могут назначаться несколько сотрудников, хотя в данной команде, всё же, определяется ведущий специалист, наиболее полно владеющий навыками написания требований, для последующей авторизации последних в Backlog’е и лично определяющий приоритеты. Роль:
   1. Формулировка требований.
   2. Приоритетность требований.
   3. Корректировка приоритетов на каждом спринте.
   4. Персональная ответственность за ценность требований для рынка/пользователей.
   5. Ответственность за взаимодействие с рынком.
2. ***Scrum Master[[9]](#footnote-9)*** - весьма сложная роль. В классических project management существует руководитель проекта. В Scrum данная роль не предусмотрена, хотя синонимом будет служить «Администратор». Scrum Master – является организатором работ команды проекта, но не вмешивается в её работу, а также следит за корректным применением Agile, и процессов (ритуалов) Scrum.
   1. Scrum Master не назначает людей на выполнение задач – это определяет команда.
   2. Scrum Master не принуждает людей делать работу – это ответственность команды.
   3. Мастер не указывает владельцу какие требования необходимо написать.
3. ***Team (Команда проекта)***, несёт ответственность за разработку продукта итерациями (спринтами). Обладая кросс-функциональностью, взаимозаменяемостью и самоорганизацией команда определят:
   1. Продолжительность спринта.
   2. Ёмкость (Capacity) команды.
   3. Размер её фокус фактора (коэффициент слаженности).
   4. Трудоёмкость требований, которые будут реализованы в спринте.
   5. Очерёдность выполнения задач.

Помимо артефактов, методология Scrum предполагает некоторые процессы, именуемые ритуалами. Каждый из процессов выполняется неукоснительно и в строгом соответствии с подходом. С практической точки зрения, не всегда применяется строгий подход, ритуалы немного адаптируются под среду.

1. ***Sprint Planning Meeting[[10]](#footnote-10)*** - сущность встречи весьма проста, но сложна по содержанию. В Startup’е проекта может занимать до 6-ти часов. В структуре представляются и поясняются требования Product Owner, задаются вопросы со стороны команды, декомпозируются требования к задачам и их оценка. Встреча по планированию спринта характеризуется:

* Выполнением всей командой перед началом спринта;
* Выбором требований из Product Backlog и формированием Sprint Backlog;
* Каждая задача проходит оценку в трудозатратах или универсальных единицах.

1. ***Daily Meeting[[11]](#footnote-11)*** - особенностью ежедневных встреч является слежение за данным процессом Scrum Master’ом, так же последний побуждает участников высказываться и слушать говорящего. В случае максимально эффективной работы, команда проводит Daily Meeting самостоятельно, обмениваясь опытом, с учётом:

* Ежедневного проведения в одно и то же время;
* Встреча проходит только стоя;
* Длительность встречи не более 15-ти минут;
* Обязательные вопросы: что я делал вчера, чем занимаюсь сегодня, какие есть проблемы?

1. ***Sprint Review[[12]](#footnote-12)*** - ценность данного процесса заключается в том, что независимо от результата пройденных работ, команда проводит демонстрацию последних определяя следующую структуру встречи:

* Зачитывание требований из Sprint Backlog;
* Приём демонстрируемых результатов по каждому критерию;
* Запись каждого вопроса со стороны Product Owner’а, для дальнейшей оценки и подготовки ответа;
* Фиксация каждого нового требования Product Owner’а, для последующего включения его в Product Backlog.

1. ***Retrospective[[13]](#footnote-13)*** - процесс характеризуется обменом опытом внутри команды. На встрече присутствует вся команда и Scrum Master, и, по необходимости, Product Owner. Методология проведения встреч вариабельна, зависит от проекта. Однако существует ряд вопросов, на которые необходимо дать ответы;

* Какие решения необходимо принять команде, чтобы процесс работ стал более предсказуемым?
* Какие проблемы нивелируют взятые на себя обязательства?
* Каким образом улучшить взаимодействие с Product Owner’ом?
* Какие ошибки совершает команда и почему?

Решения фиксируются на каждой доске, и после всеобщего голосования решения подлежат исполнению со следующего спринта. Scrum Master’ом контролируется ход встречи и соответствие её регламенту.

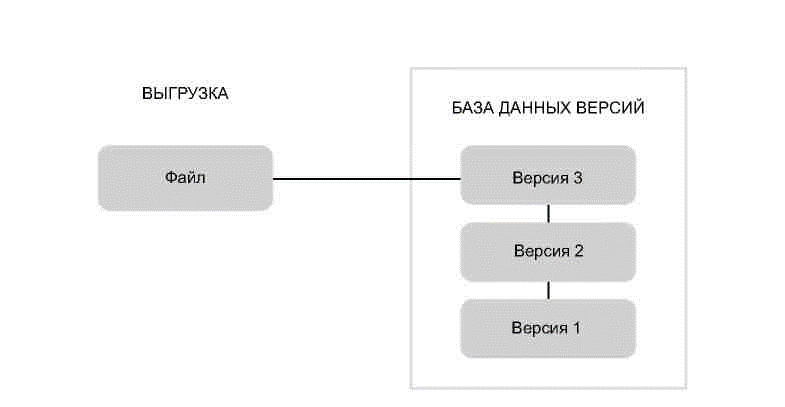
1. **Репозиторий Git, система управления версиями**

Репозиторий – происходит от английского repository, и дословно означает хранилище, место, где хранятся и поддерживаются какие-либо данные. В нашем случае – это хранилище файлов программных продуктов. Репозиторий Git весьма популярная система контроля версий и совместной разработки проектов с исходным открытым кодом. Задумка и создание проекта принадлежит Линусу Торвальдсу, была реализована в управлении ядром операционной системы Linux. Что же на самом деле «управление версиями» или система управления версиями на современном этапе? Наиболее оптимальное понятие стоит рассматривать с точки зрения записей истории изменений файла, либо набора файлов, для дальнейшей возможности возврата к конкретным версиям исходных данных. Яркий пример приводится Scott Chacon и Ben Straub: Хранение всех версий или компоновок графическими, или веб-дизайнерами с использованием системы контроля версий (Version Control System, VCS). Данная система позволяет возвращать в предыдущие состояния как отдельных файлов, так и целого проекта, сравнивать сделанные изменения, отслеживать, кто и когда последним редактировал файл. Существует три варианта системы контроля версиями:

1. ***Локальные системы контроля версий***

В качестве простого примера метода контроля данных локальных версий можно привести действия пользователей по элементарному копированию файлов в другую папку (иногда с метками времени) и возможностью синхронизации (портфель Windows). Однако, регулирование данного вопроса с позиции программистов, реализовано локальной системой контроля версий, оснащённой простой базой данных, хранящих всю информацию об изменениях файлов.

Изображение № 2 Локальное управление версиями

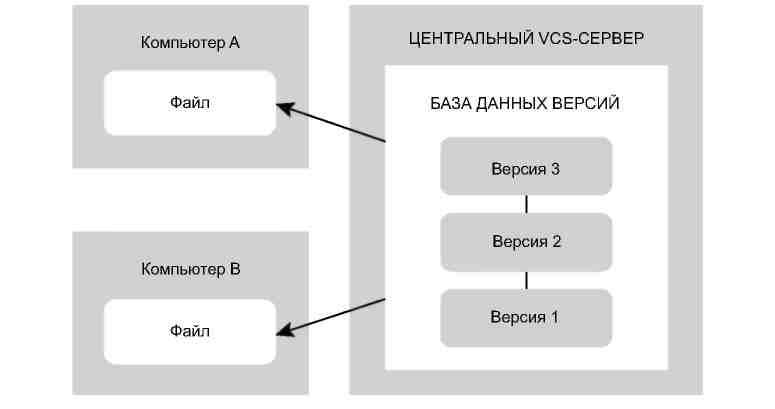


Система конроля редакций (Revision Control System, RCS), одна их популярных локальных систем контроля версий. В настоящее время используется в операционной системе Mac OS X.

1. ***Централизованные системы контроля версий***

Одной из существенных проблем для разработчиков программных продуктов, действующих в различных системах – взаимодействие. Решением послужило внедрение централизованных систем контроля версий (Centralized Version Control System, CVCS), представленных Subversion и Perforce. Данные системы обладают единым сервером, содержащим все версии файлов, и набором клиентов, выгружающих файлы с сервера. Долгое время подобный подход сохранял лидирующие позиции и был стандартом управления версиями.

Изображение № 3 Централизованное управление версиями

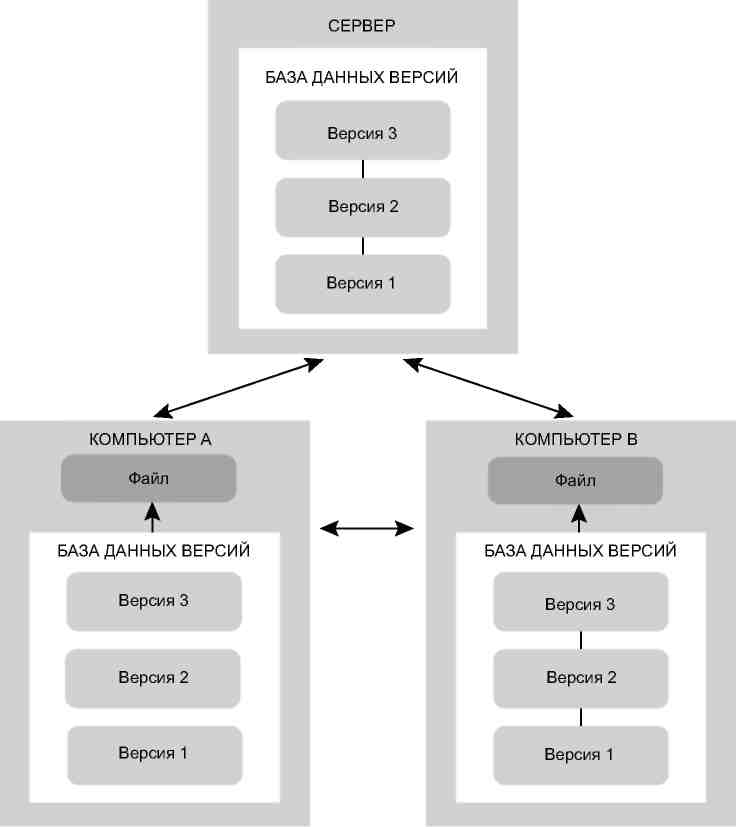


Данная схема имеет значительные преимущества перед локальными системами контроля версий. Безусловно, каждый сотрудник, работающий над проектом, в определённой степени знает, что делают его коллеги. Также существует возможность контролировать права доступа различными сотрудниками, администрирование CVCS намного проще, хотя у подобной схемы есть существенный недостаток, лежащий на поверхности – точка отказа, представленная центральным сервером. И самое главное, невозможность работы локально, при отказе сервера и отсутствии резервных копий информации у клиента.

1. ***Распределённые системы контроля версий (Distributed Version Control System)***

В DVCS (Git, Mercurial, Bazaar или Darcs) клиенты, выгружая последние снимки файлов, создают при этом зеркальную копию репозитория. И, таким образом, при выходе из строя одного из серверов, его работоспособность восстанавливается одним из клиентских репозиториев. Каждая выгрузка сопровождается полным резервным копированием всех данных.

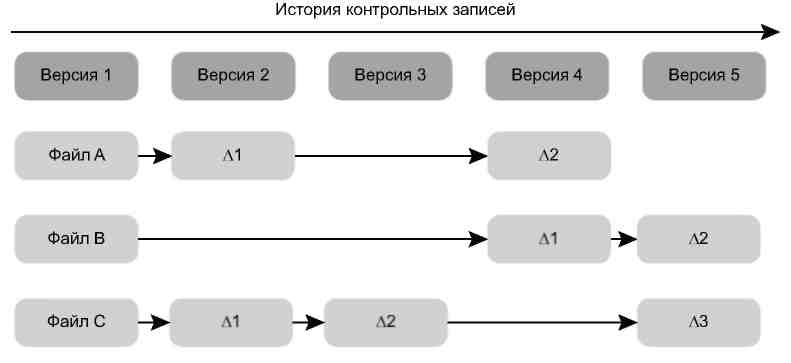
Изображение № 4 Распределённое управление версиями



Помимо этого, данная система обладает удалёнными репозиториями, что позволяет разным группам сотрудничать в рамках одного проекта, а также использовать методологию Scrum дистанционно.

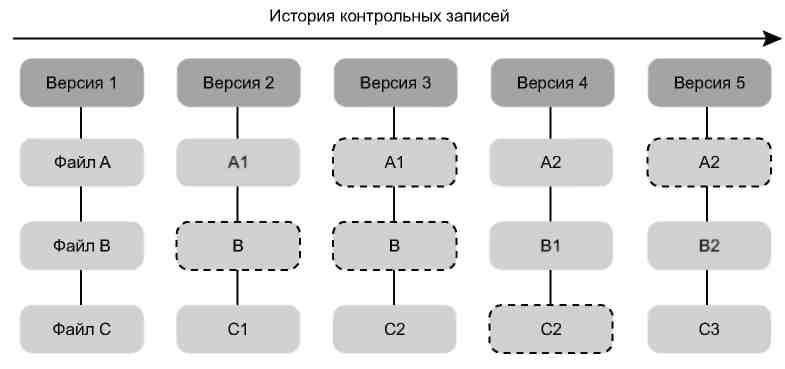
Стоит отметить, что одним из главных отличий Git от прочих систем контроля версий является восприятие данных. Многие системы хранят информацию в виде списка изменений, связанных с файлами. Подобные системы (CVS, Subversion, Perforce, Bazaar), рассматривают хранимые данные, как набор файлов и изменений, вносимых в эти файлы на протяжении их жизненного цикла.

Изображение № 5 Хранение данных в виде изменений



С позиции Git, данные представляют собой набор снимков состояния миниатюрной файловой системы. Создавая новую версию или сохраняя состояние проекта в Git, происходит снимок всех файлов в конкретный момент времени, что повышает продуктивность работ. Git воспринимает данные как поток снимков состояния (stream of snapshots).

Изображение № 6 Хранение данных в виде снимков состояния проекта



Ещё одной особенностью Git является локальность операций, то есть Git требуются только локальные файлы и ресурсы. К примеру, просмотр истории проекта происходит чтением из локальной базы данных, и таким образом, историю видно сразу. При необходимости и наличие сети данные синхронизируются с сервером, что можно подтвердить механизмом вычисления контрольных сумм для сохранения целостности Git. Механизм основан на хеше SHA-1 (Secure Hash Algorithm[[14]](#footnote-14)), строки из 40-а символов, включающей в себя числа в шестнадцатеричной системе (0-9 и a-f) и вычисляемой на основе содержимого файла или структуры папки в Git. Таким образом, практически все операции в Git приводят к добавлению данных в базу, что отражает снимок состояния проекта, а не отдельных файлов, которые весьма просто потерять. Это, в свою очередь, отражает три состояния Git:

* ***Зафиксированное (committed)*** – означает сохранение данных в локальной базе данных.
* ***Модифицированное (modified)*** – означает, что изменения уже внесены в файл, но пока не зафиксированы в базе данных.
* ***Индексированное (stage)*** – пометка текущей версии модифицированного файла предназначенную для последующей фиксации.

В результате Git-проект разбивается на три основные области: папка Git, рабочая папка и область индексирования.

Изображение № 7 Рабочая папка, область индексирования и папка Git



* ***Папка Git*** – место, где хранятся метаданные и объектная база данных проекта.
* ***Рабочая папка*** – место, куда выполняется выгрузка одной из версий проекта.
* ***Область индексирования*** – файл, расположенный в папке Git и хранящий информацию о том, что именно войдёт в следующую операцию фиксации. В некоторых случаях данную область называют промежуточной.

Рабочий процесс Git:

1. Редактирование файлов в рабочей папке.
2. Индексирование файлов с добавлением снимков последних в область индексирования.
3. Фиксация файлов из области индексирования снимками в папке Git.

Если существует конкретная версия файла в папке Git, то файл считается зафиксированным, соответственно, внося изменения в файл, он переноситься в область индексирования. Отредактированный файл после выгрузки считается модифицированным.

В создании репозитория Git-проекта существует два подхода, использование существующего проекта или папки, и клонирование существующего репозитория с другого сервера. Инициализация репозитория в существующей папке осуществляется командой:

1. *$ git init*

В результате которой, в существующей папке появляется ещё одна папка с именем .git и всеми необходимыми нам файлами репозитория – это основа Git-репозитория. Стоит отметить, что на данном этапе отсутствует контроль файлов. Для этого потребуется указать файлы, за которыми должна следить система, то есть выполнить фиксацию изменений. Для это потребуется несколько команд git add, добавляющих файлы, а затем команда git commit (непосредственно фиксация):

1. *$ git add \*.c*
2. *$ git add LICENSE*
3. *$ git commit –m ‘версия проекта’*

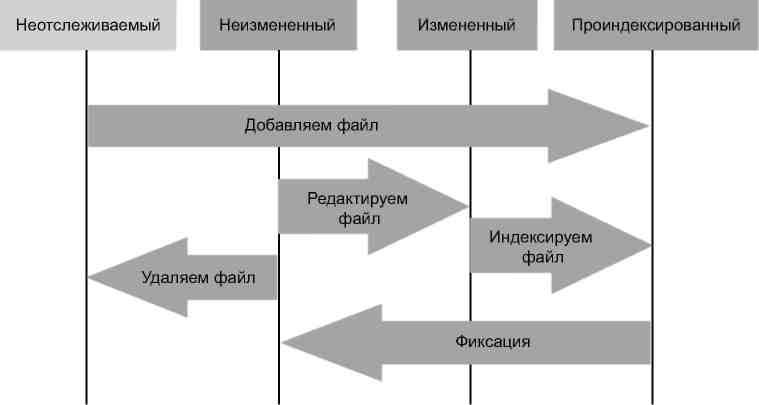
Иногда необходимо принять участие в каком-либо проекте, для этого служит команда git clone. Особенность данной команда в разрезе Git, является то, что последняя забирает все версии всех файлов за всю историю проекта. Например, клонирование подключаемой Git-библиотеки libgit2:

1. *$ git clone* [*https://github.com/libgit2/libgit2*](https://github.com/libgit2/libgit2)

Будет создана папка с именем libgit2, в ней инициализируется папка .git, считывающая и выгружая все данные из репозитория. Для работы Git применяются различные транспортные протоколы, они будут рассмотрены во второй главе.

Проведя вышеперечисленные действия, был создан или клонирован репозиторий. Выполняя дальнейшую работу, необходимо вносить и фиксировать записи изменений в репозиторий. Необходимо помнить, что каждый файл пребывает в двух состояниях – отслеживаемом и не отслеживаемом. В первом случае – файлы, входящие в последний снимок системы; могут быть неизменёнными, изменёнными и подготовленными к фиксации. Во втором случае – файлы рабочей папки, не вошедшие в последний снимок системы и не проиндексированные для последующей фиксации. Однако первое клонирование репозитория делает файлы отслеживаемыми и неизменёнными. Отредактированный файл Git рассматривает как изменённый, его состояние отличается от последнего зафиксированного. Индексируя данные файлы, фиксируются все изменения, далее цикл повторяется.

Изображение № 8 Жизненный цикл состояния файлов



Проиндексировав файлы, необходимо определить состояние файлов. Выполнить это можно используя команды:

1. *$ git status*

* *On branch master[[15]](#footnote-15)*
* *Nothing to commit, working directory clean*

Вывод результата означает отсутствие отслеживаемых и изменённых файлов, и соответственно отсутствие ветвления. Простым примеров демонстрации служит файл README, предлагаемый GitHub’ом по умолчанию. Добавляя его, получаем не отслеживаемый файл:

1. *$ echo ‘Name project’ > README*
2. *$ git status*

* *On branch master*
* *Untracked files: (use “git add <file>…” to include in what will be committed)*
* *README*
* *nothing added to commit but untracked file present (use “git add” to track)*

Untracked file даёт нам понять, что новый файл README является не отслеживаемым, данное состояние, указывает на видимость данного файла, но его отсутствие в предыдущей фиксации (коммите). Каким же образом его можно отследить? Для этого используется команда git add, для файла README она будет выглядеть следующим образом:

1. *git add README $ git status*

* *On branch master*
* *Changes to be committed: (user “git reset HEAD <file>… : to unstage)*
* *New file: README*

Таким образом, все проиндексированные файлы перечисляются под заголовком Change to be committed. Если в этот момент произвести фиксацию, версия файла, существовавшая на момент выполнения команды git add, попадает в историю снимков состояния. Произвести индексацию файлов, находящихся под наблюдением, в свою очередь можно командой git.add и затем запустить команду git.status.

1. *$ git add \*.rb (\* - любой модифицируемый файл)*
2. *$ git stutus*

* *On branch master*
* *Changes to be committed: (use “git reset HEAD <file>…” to unstage)*
* *New file: README*
* *modified: \*.rb*

Теперь оба файла проиндексированы и войдут в следующий коммит. Однако, если внести изменения в файл и выполнить команду git status, можно обнаружить следующее:

1. *$ vim \*.rb*
2. *$ git status*

* *On branch master*
* *Changes to be committed: (use “git reset HEAD <file>…” to unstage)*
* *New file: README*
* *modified: \*.rb*
* *Changes not staged for commit:*
* *(use “git add <file>…” to update what will be committed)*
* *(use “git checkout -- <file>…” to discard changes in working directory)*
* *Modified: \*.rb*

Данные значения команды определяют, что файл \*.rb одновременно является и проиндексированным, и не проиндексированным. Это означает, что команда git add требует повторного выполнения. Ещё одной особенность команды git status служат флаги[[16]](#footnote-16), позволяющие получить сведения в более компактной форме. Флаг –s или --short позволят вывести файлы с дополнительными метками в краткой форме:

* Метка А – файлы, добавленные в область предварительной подготовки;
* Метка М – модифицированные файлы
* Метка ММ – индексирован и модифицирован;
* Метка ?? – новые, не отслеживаемые файлы.

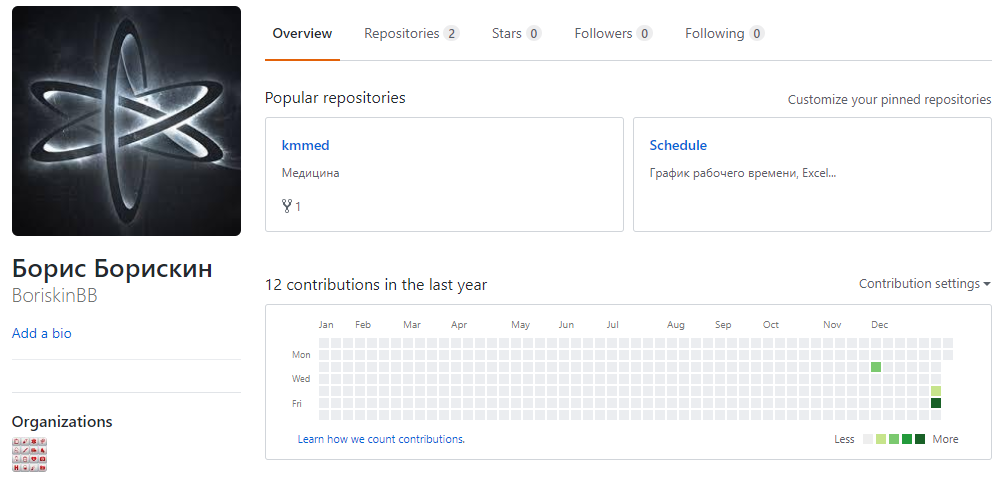
# **Глава 2 Создание публичного репозитория GitHub, принципы и требования**

1. **Public GitHub в хостинг приложении, создание, протоколы**

Сайт GitHub является одним из крупнейших из хостингов для Git-репозиториев и центральным местом сотрудничества большого количества разработчиков и проектов. Именно на этом сайте располагается подавляющая часть всех Git-репозиториев, а многие проекты с открытым исходным кодом используют его для хостинга Git, отслеживания ошибок и рецензирования кода. Стоит отметить, что учётные записи, предоставляемые на безвозмездной основе, имеют всю функциональность, но при этом все проекты будут полностью открытыми. Логотип GitHub’а – гибрид кота и осьминога (Осьмикот).

Автором данной работы создана учётная запись на странице https:\\github.com. Сайт имеет англоязычную версию.

Изображение № 9 Профиль аккаунта GitHub



Протокол подключения к Git-репозиториям https:// protocol, либо настройка ключа SSH. Всего же, для передачи данных Git использует четыре протокола: локальный, HTTP, SSH (Secure Shell) и Git.

1. ***Локальный протокол (Local Protocol)*** – основной, используется в случаях, когда роль удалённого репозитория играет папка на локальном диске клиента. Клонирование осуществляется командой, с указанием пути в виде адреса URL[[17]](#footnote-17):
2. *$ git clone /opt/git/project.git*

Либо:

1. *$ git clone* [*file:///opt/git/project.git*](file:///opt/git/project.git)

Преимущество данного протокола заключается в том, что, используя полномочия общего группового доступа к папке можно работать командой, однако, находясь вне локальной сети настройка общего сетевого диска вызовет определённые сложности.

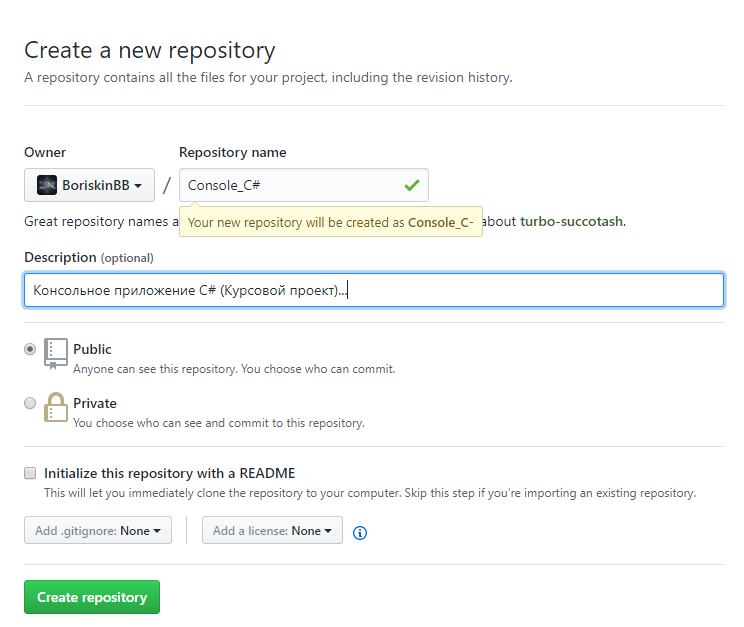
1. ***Протоколы HTTP[[18]](#footnote-18)*** - существуют в двух режимах:
   1. Простой протокол http – был единственным до версии Git 1.6.6, в настоящее время используется в случае отсутствия реакции сервера на интеллектуальный протокол HTTP, настройка осуществляется путём размещения пустого репозитория в корневую папку с HTTP-документами и настройка хука (hooking – технология, изменяющая стандартное поведение тех или иных компонентов информационной системы, перехват), срабатывающего после обновления (post update hook). Предоставление доступа на чтение по протоколу:
2. *$ cd /vfr/www/htdocs/*
3. *$ git clone --bare /path/to/git\_project gitproject.git*
4. *$ cd gitproject.git*
5. *$ mv hooks/post-update.sample hooks/post-update*
6. *Chmod a+x hooks/post-update*
   1. Интеллектуальный протокол http – функционирует аналогично протоколу SSH или Git, но передаёт данные через стандартные порты HTTP/S. Простота работы заключается в возможности аутентификации по имени пользователя/паролю (избежание настройки ключей SSH).
7. ***Протоколы SSH[[19]](#footnote-19)*** - протокол с проверкой подлинности, настроен на многих серверах, для клонирования Git-репозитория указывается префикс ssh://URL:
8. *$ git clone ssh://user@server: project.git*

Достоинство данного протокола в простоте настройки – SHH-демоны (программы в системах класса UNIX, запускаемые в фоновом режиме) встречаются повсеместно, и все данные передаются в зашифрованном виде. Однако протокол SSH не позволяет анонимный доступ к репозиторию.

1. ***Протокол Git*** – поставляется со специальным демоном, который слушает порт 9418. Данный демон предоставляет сервис, в целом напоминающий протокол SSH, но без необходимости аутентификации. Для работы репозитория по протоколу Git необходимо создать файл git-export-daemon-ok, без него демон обслуживать репозиторий не станет. В данной ситуации страдает безопасность, репозиторий Git может быть доступен для клонирования или всем, или никому. Как правило, Git протокол является самым доступным. В случае большого проекта с открытым исходным кодом данный протокол окажется самым быстрым из-за отсутствия шифрования и аутентификации.

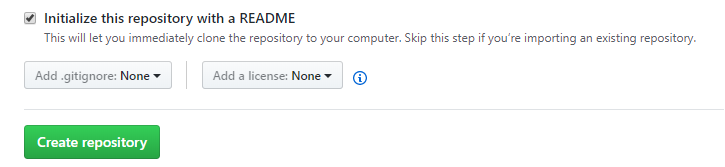
Создание нового репозитория в GitHub не столь затруднительно как кажется:

Изображение № 10 Создание репозитория консольного приложения



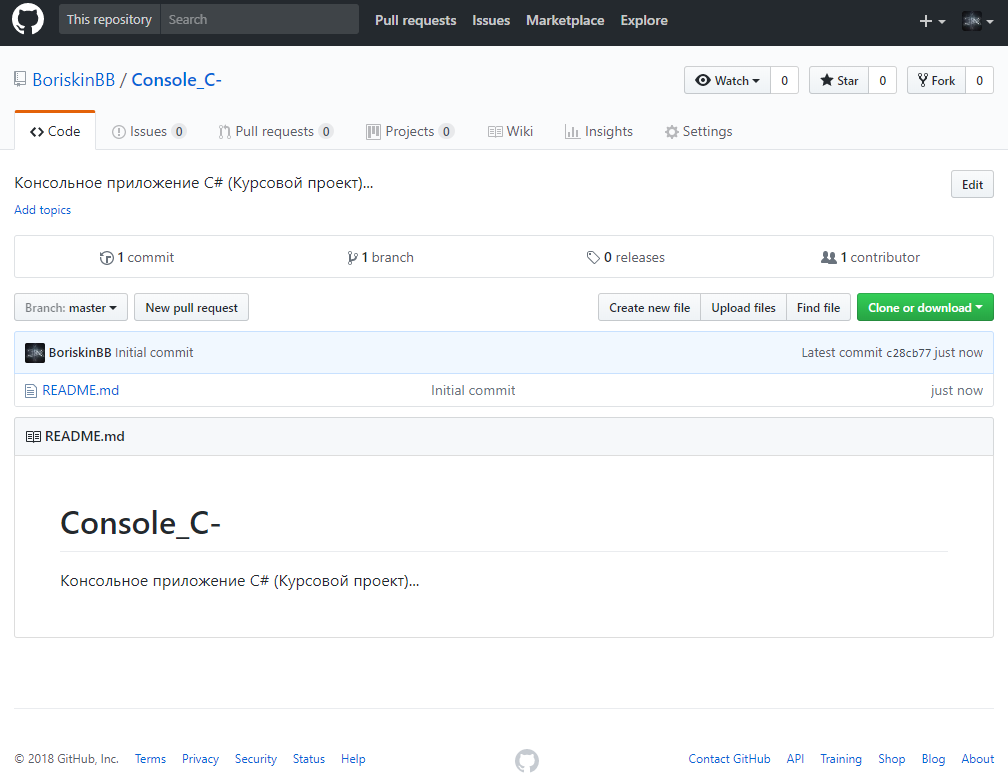
К созданию данного репозитория, автоматически предлагается инициализация его с помощью файла README:

Изображение № 11 Инициализация файла README



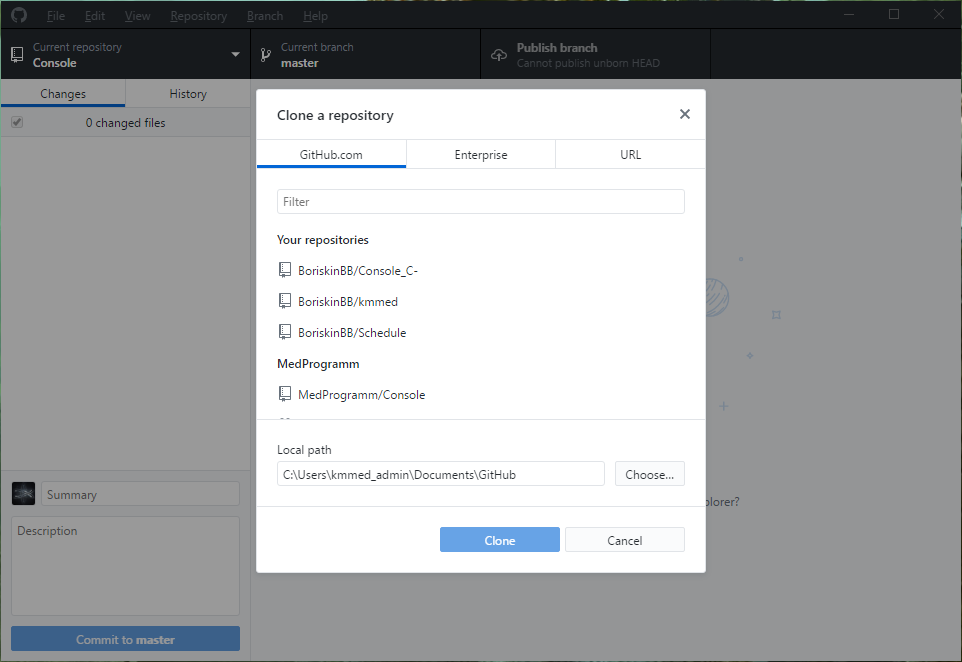
Преимуществом данного действия будет являться возможность автоматического клонирования репозитория в GitHub Desktop (Клиентской версии).

Изображение № 12 Созданный репозиторий



В результате проведённых действий: одна фиксация (commit) и одна ветвь (branch), а также клонирование созданного репозитория в настольную версию. С указанием локального пути репозитория. Следует заметить, что GitHub избирателен к имени репозитория, замена C# на С-.

Изображение № 13 Клонирование репозитория в настольную версию



1. **GitHub Desktop, Git Bash: Основы работы**

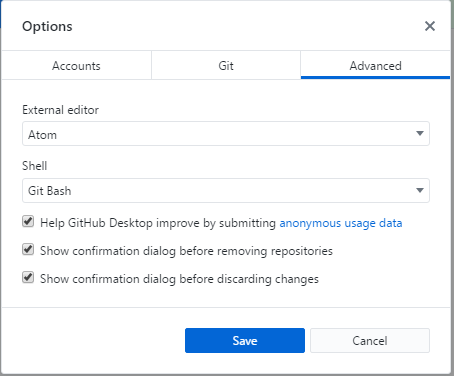
Прежде чем рассмотреть основы работы в настольной версии GitHub и командной строки Git Bash, предлагаемой в настройках настольной версии, стоит ещё раз зафиксировать основные команды Git:

1. ***Push (продвинуть, выдвинуть от себя)*** – отправка изменений из локального репозитория в удалённый репозиторий.
2. ***Fetch (получать, выбирать)*** – получение изменений из удалённого репозитория для сравнения и возможного последующего слияния.
3. ***Merge (идти)*** – слияние, применение изменений совершённых в другом репозитории текущим репозиторием.
4. ***Pull (тянуть, вытаскивать)*** – комбинация fetching и merging. Получение из удалённого репозитория списка изменений и применение их к текущему репозиторию.

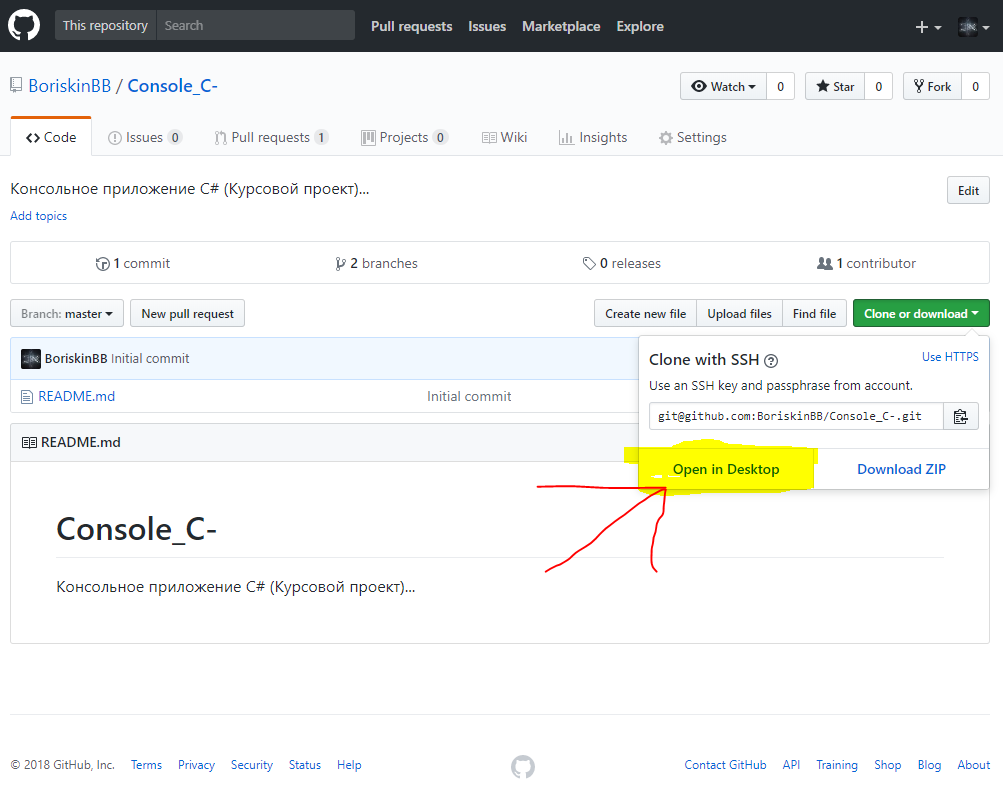
Рассматривая клиентскую версию GitHub’а стоит отметить возможность графической работы с репозиторием, хотя в опциях существует возможность работы для продвинутых пользователей, при чём можно установить внешний редактор (External editor), а также выбрать оболочку (Shell) командной строки (Изображение № 14). В нашем случае, внешним редактором установлен Atom, а командной строкой выбран Git Bash. В настольную версию были клонированы все репозитории имеющиеся на <https://github.com/BoriskinBB>, при чём web-приложение даёт возможность открыть репозиторий в настольной версии, не прибегая к командной строке (Изображение № 15). Клонировав репозитории, получаем локальную рабочую версию. Её можно использовать без подключения к сети. Синхронизацию с сервером можно провести позднее.

Основой работы в Git является ветвление (branching), то есть отклонение от основной линии разработки. Настольная версия позволяет создавать новую ветку (Изображение № 16), как с использованием интерфейса, так и при помощи командной строки Git Bash. Команда *$ git help*, позволяет увидеть все функции командной строки (Изображение № 17).

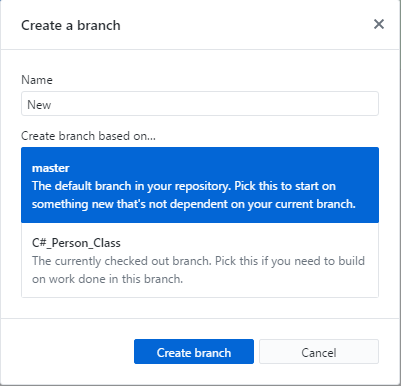
Изображение № 14 Опции GitHub Desktop



Изображение № 15 Клонирование репозитория



Изображение № 16 Создание новой ветки в репозитории



Изображение № 17 Параметры командной строки Git Bash



1. **Распределённый GitHub в консольных версиях C# Visual Studio, Scrum методология**

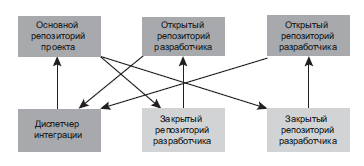
Основной чертой распределённых рабочих процессов в системе Git, в отличии от прочих централизованных систем контроля версий, это возможности более гибкой совместной работы над проектами. В централизованных системах, разработчики представлены как узлы сети, производящие работу на центральном концентраторе. В Git каждый из разработчиков может выступать как в роли узла, так и в роли концентратора. Таким образом, каждый из участников может вносить код в чужие репозитории и поддерживать открытый репозиторий. Данный подход расширяет поле деятельности команды.

Из предыдущих глав данной работы, касающихся Scrum методологии, мы знаем, что над проектом работает команда. Каким же образом данная методология может быть применима в GitHub’е? Пожалуй, в первую очередь стоит рассмотреть роли в команде разработчиков. Подобные роли используются в довольно крупных проектах, однако, можно провести аналогию со Scrum.

Одной из разновидностей рабочего процесса в GitHub, является работа с набором репозиториев (условно – задач в Scrum), где участники разделены на диктатора и помощников (диспетчеров интеграции). Диктатор является главенствующим диспетчером интеграции. Но прежде чем говорить о данном процессе, стоит рассмотреть общую последовательность действий диспетчеров интеграции при обычной работе с общим репозиторием (официальным проектом):

1. Владелец проекта фиксирует файлы в общий репозиторий.
2. Участники проекта клонируют этот репозиторий и вносят свои изменения.
3. Каждый из участников выкладывает изменения в свой открытый репозиторий.
4. Участники отправляют владельцу письмо с просьбой включить его изменения в общий проект.
5. Владелец добавляет репозиторий каждого участника как удалённый и выполняет локальное слияние.
6. Владелец отправляет слитые изменения в основной репозиторий.

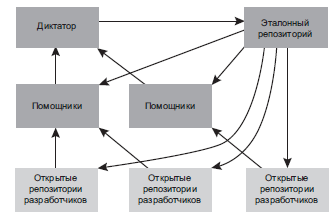
Изображение № 18 Рабочий процесс диспетчера интеграции



Преимуществом данного подхода является возможность продолжать работу, не зависимо от основного владельца репозитория, и в своём темпе. Рабочий же процесс с диктатором выглядит следующим образом:

1. Разработчики производят работы в тематических ветках и перемещают результаты своей работы на вершину ветки ***master***. Ветка принадлежит диктатору.
2. Помощники сливают тематические ветки разработчиков в свои ветки ***master***.
3. Диктатор сливает содержимое веток ***master*** своих помощников в свою ветку ***master***.
4. Диктатор отправляет свою ветку ***master*** в эталонный репозиторий, что позволяет остальным разработчикам возможность перемещать туда свои данные.

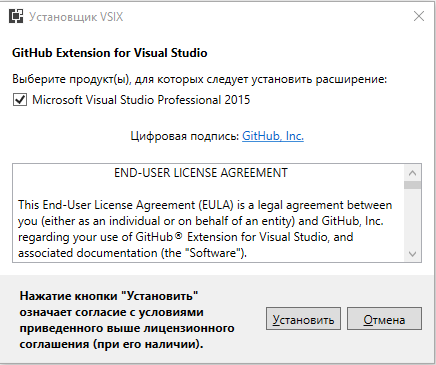
Изображение № 19 Рабочий процесс с диктатором и помощниками



Подобный процесс, нечастое явление, однако он может пригодиться в больших проектах, где необходимо строгое иерархическое окружение. Делегируя полномочия разработчикам, мы в целом получаем Scrum Master из диктатора.

Каким же образом может выглядеть распределённый GitHub в консольных версиях приложений C# Visual Studio? Прежде всего необходимо установить расширение GitHub Extension[[20]](#footnote-20) for Visual Studio.

Изображение № 20 Установщик GitHub Extension for Visual Studio

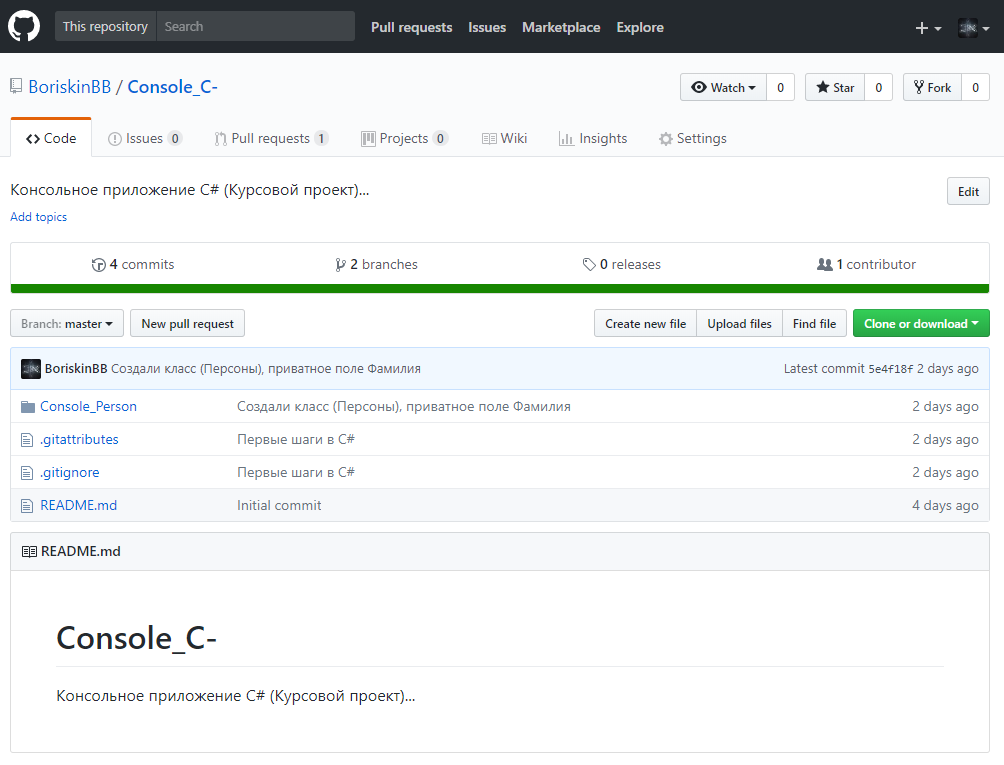


Стоит отметить, что использование данного расширения весьма эффективно расширяет и упрощает возможности работы с GitHub. Работа становиться очевидна, а синхронизация происходит непосредственно на сервере и в локальном репозитории. Помимо этого, непосредственно в данном репозитории можно создать решение. В нашей ситуации, консольным решением будет ***Console\_Person.sln***. Решение предполагает вывод в консоли персональных данных гражданина (персоны) со следующими характеристиками:

* Фамилия
* Имя
* Отчество
* Дата рождения

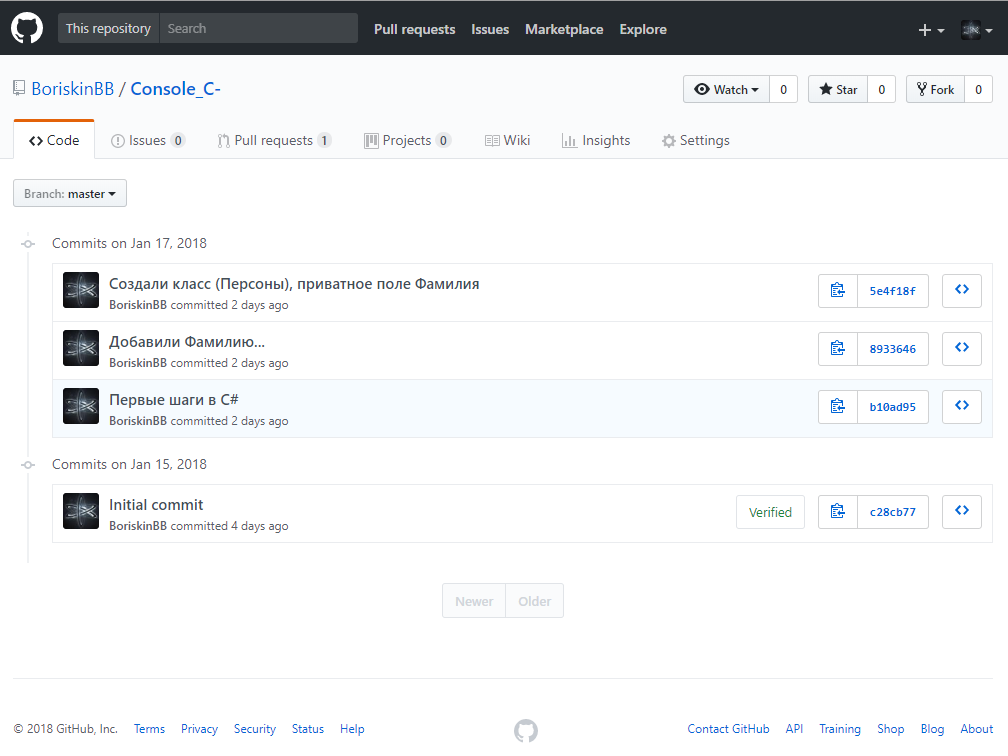
Все вносимые изменения будут фиксированы в GitHub. Далее планируется добавить поле СНИЛС (страховой номер лицевого счёта обязательного пенсионного страхования), который в последствии будет являться однозначным идентификатором персоны в последующем развитии проекта. Однако для данного кода будет создана отдельная ветка. Проект был создан ранее, и на текущий момент имеет четыре коммита (фиксации) и две ветки (одна из ветвей создана для файла README).

Изображение № 19 Репозиторий Console\_C-

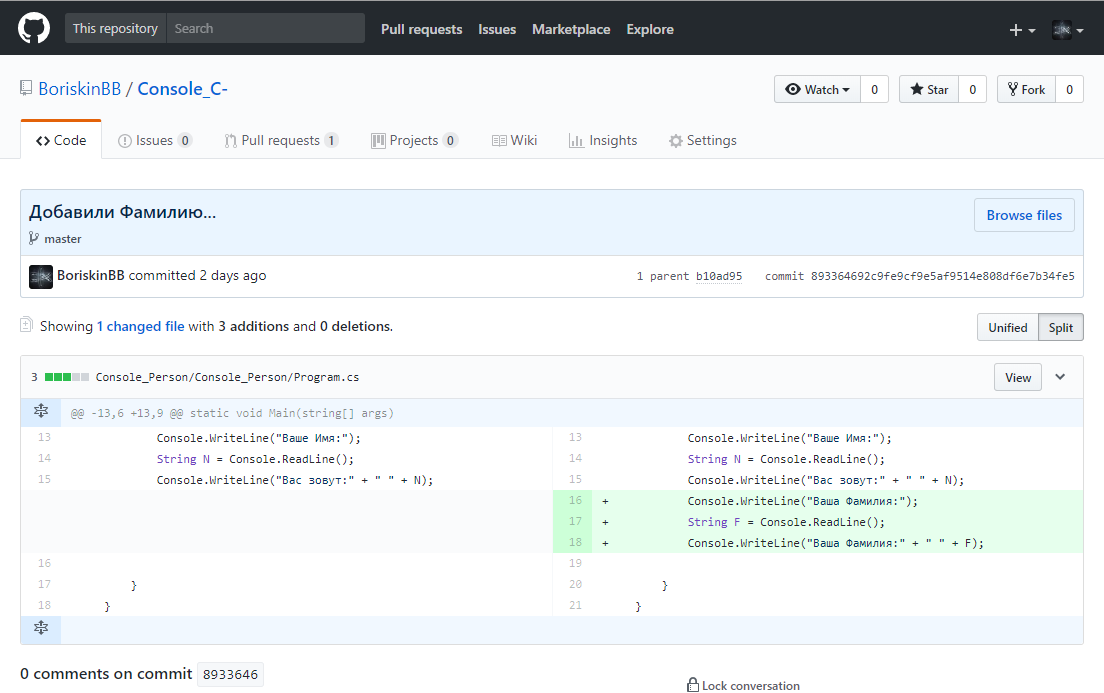


Фиксации, проведённые 17.01.2018 года отображены на вкладке commits (Изображение № 20). Произведя переходы по фиксациям можно увидеть все внесённые изменения (Изображение № 21). Дальнейшие переходы можно видеть в changed files (изменённых файлах) и как говорилось ранее были сделаны слепки состояний из всего проекта, то есть все файлы. Ещё одним важным фактором фиксаций, является то, что в ней отображены исключительно те файлы, которые подверглись изменениям и именно на период фиксации. Таким образом, проводя работы над проектом и ежедневно фиксируя состояния можно отслеживать все производимые действия.

Изображение № 20 Фиксация от 17.01.2018 года

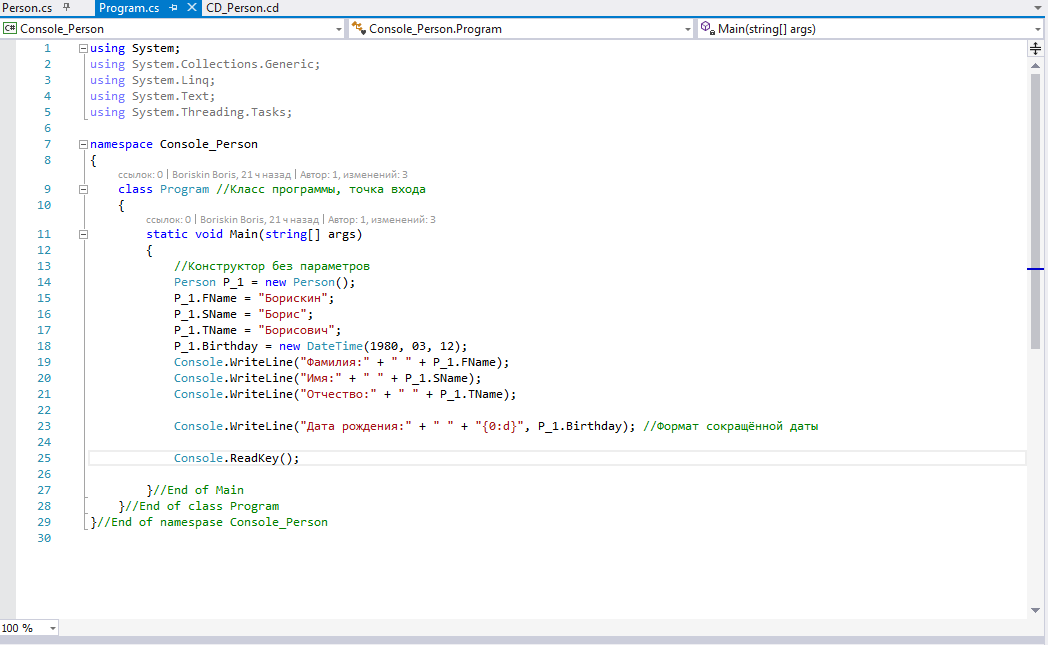


Изображение № 21 График внесённых изменений GitHub



Следующим этапом работы будет написание кода для ветки мастер и его последующая фиксация в репозитории GitHub.

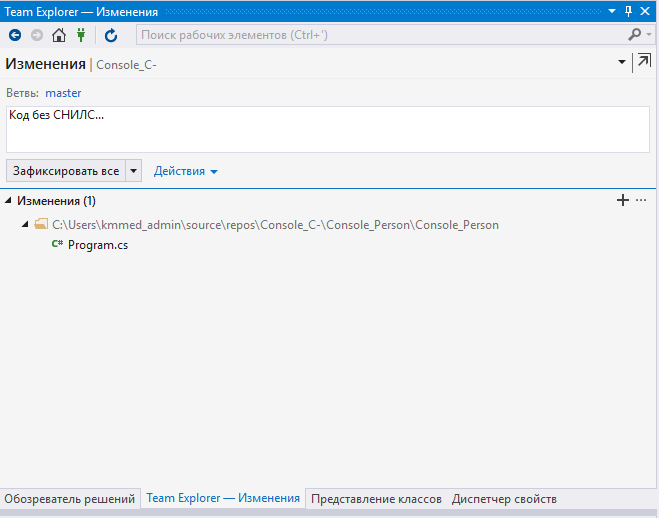
Изображение № 22 Исполняемый код ветки ***master в Visual Studio***



На текущий момент, фиксация не производилась. Проведём фиксацию в ветке ***master*** при помощи GitHub for Visual. Стоит отметить, что фиксация требует обязательного сообщения, которое в последующем будет заголовком коммита. В нашем случае обозначим данную фиксацию как код без СНИЛС. Варианты фиксации (в GitHub Desktop – команда Pull):

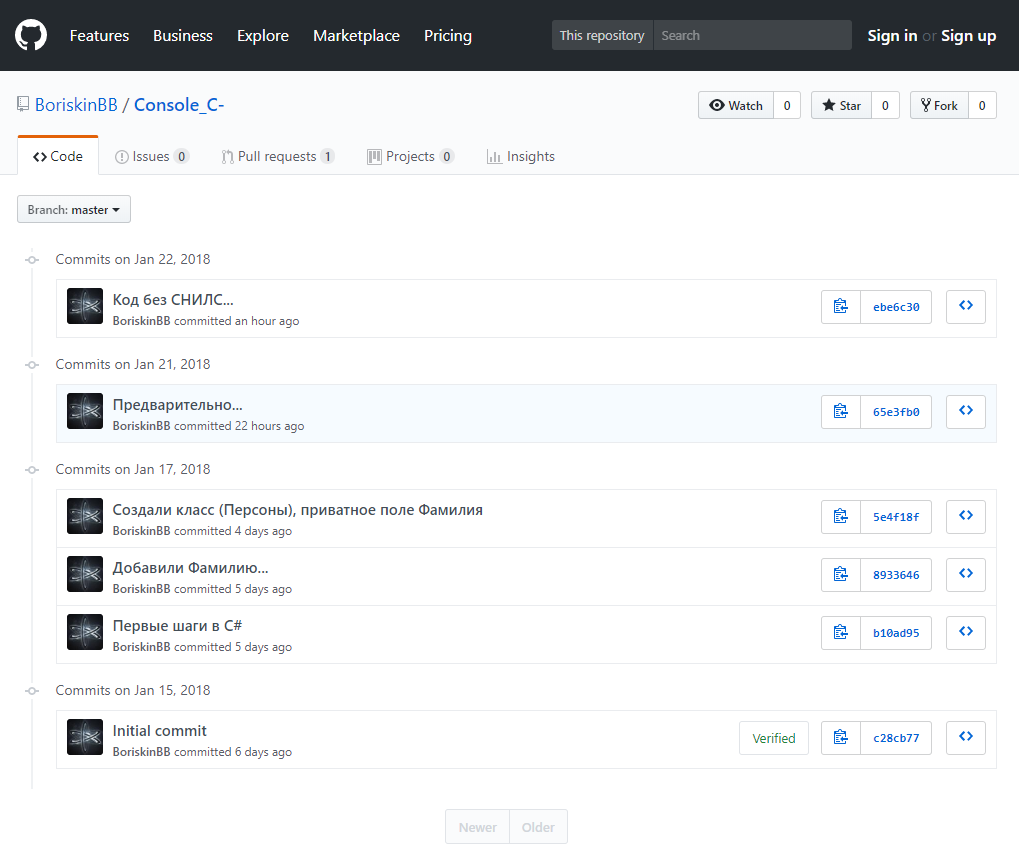
* Зафиксировать всё
* Зафиксировать всё и отправить
* Зафиксировать всё и синхронизировать (используемый вариант)

Изображение № 23 Фиксация кода в GitHub

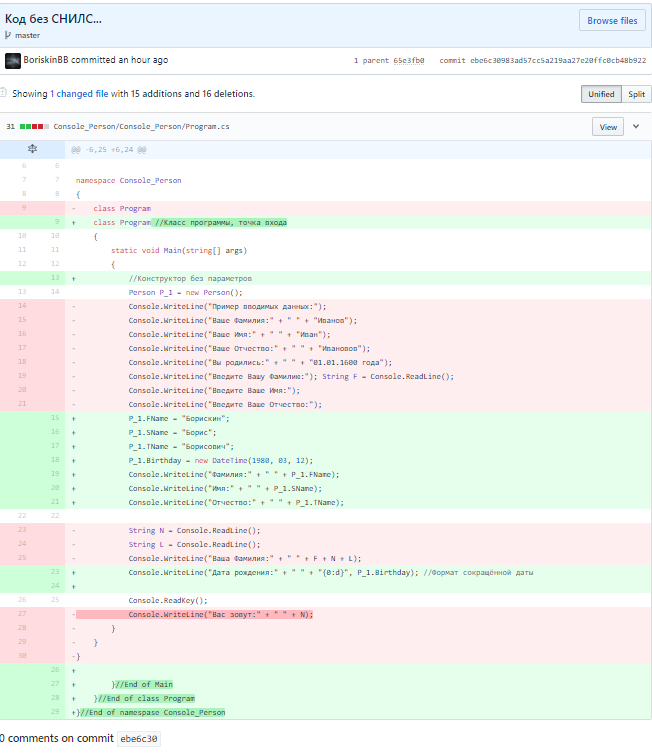


Фиксация на хостинге GitHub выглядит следующим образом:

Изображение № 24 Коммиты ветки мастер

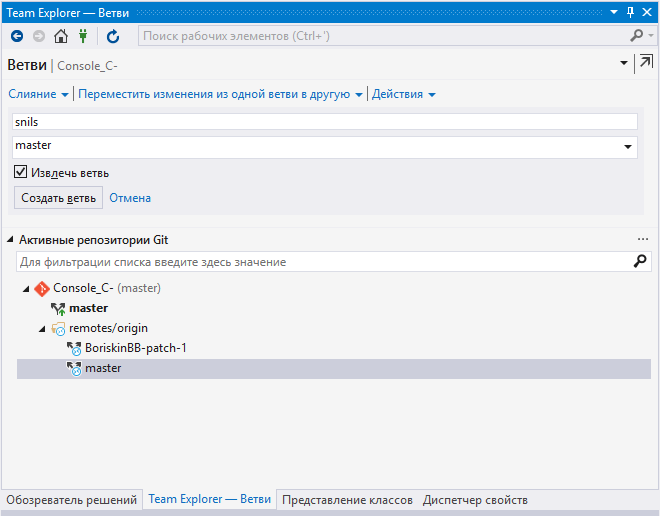


Изображение № 25 Вид изменённого файла



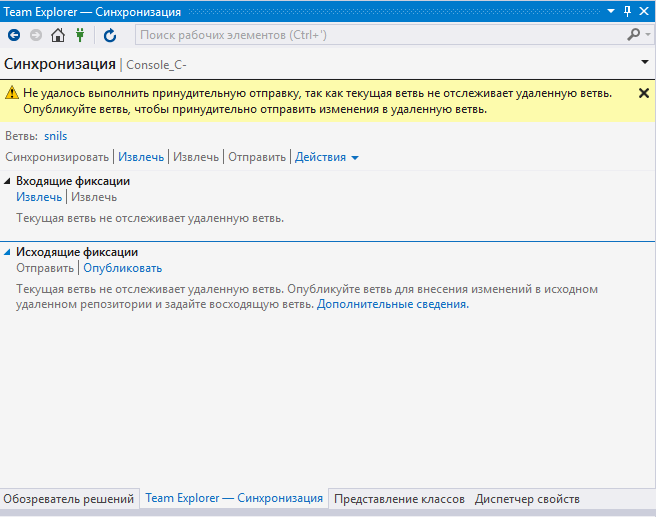
Следующим этапом внесём изменения в код, добавим поле СНИЛС. Длина строки СНИЛС равна 14, с учётом дефисов. Формат СНИЛС имеет вид: ХХХ-ХХХ-ХХХ-YY, где Х – непосредственно номер, а Y – контрольное число, вычисляемое по определённому алгоритму. Особенностью СНИЛС является его уникальность принадлежности одному человеку (гражданину). Аналогом в США служит карточка соцстрахования. В нашем случае, создав поле СНИЛС, дополнительно создадим ветку с тем же именем, в ветке ***master***. Будет использован GitHub for Visual.

Изображение № 26 Создание локальной ветки Git

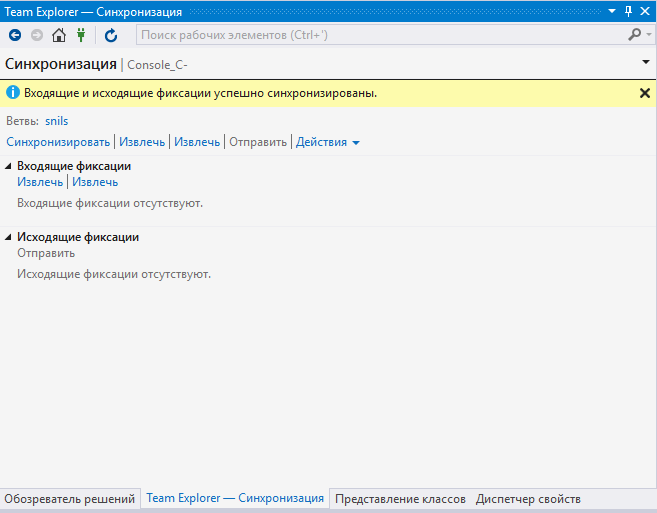


Далее проведём фиксацию кода со СНИЛС. Однако, прежде чем провести фиксацию, ветку необходимо опубликовать. Visual Studio предлагает данную операцию в интерфейсе, с информационным замечанием, что ткущая ветвь не отслеживает удалённую ветвь.

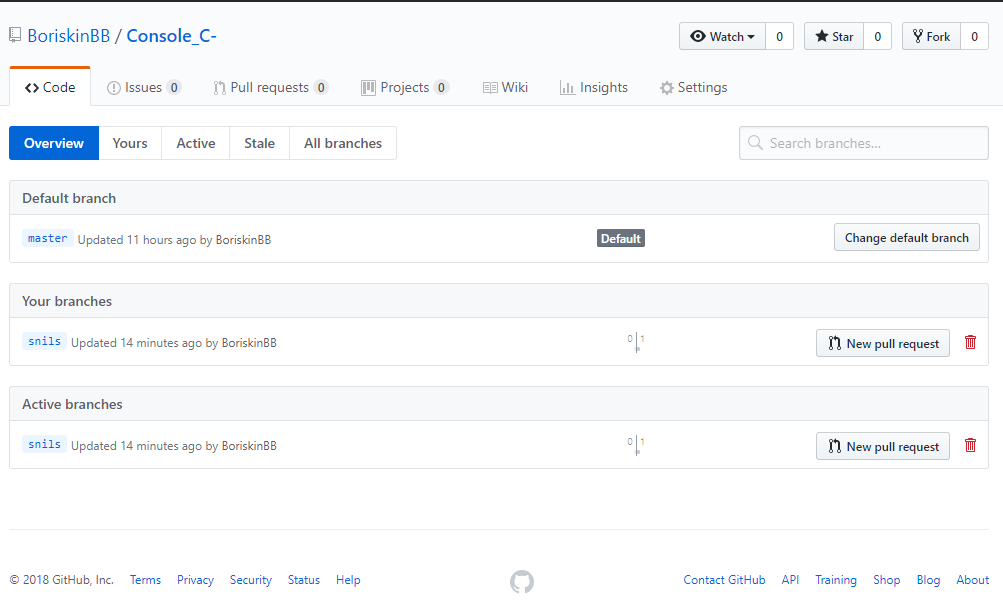
Изображение № 27 Публикация исходящих фиксаций



Изображение № 28 Фиксация/синхронизация ветви snils

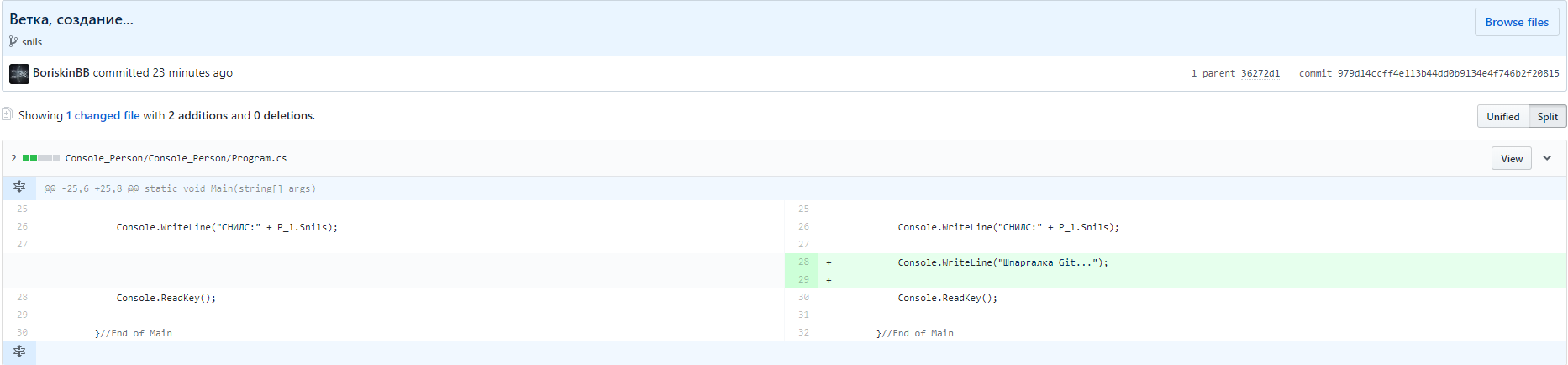


Изображение № 29 Ветвь в GitHub



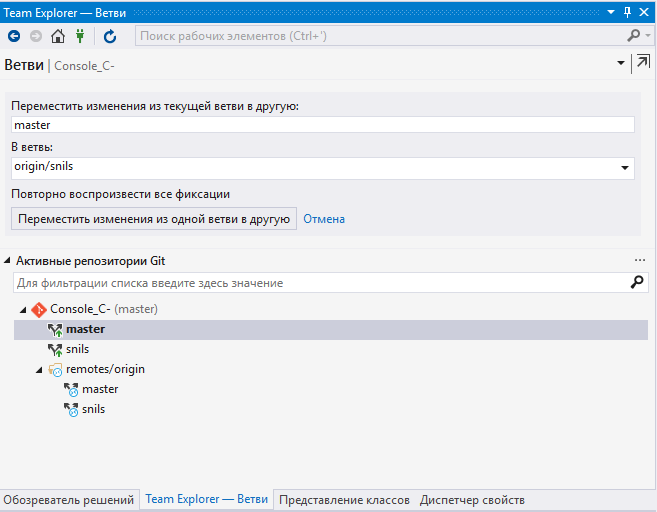
Однако стоит обратить внимание, на график изменённых файлов, в данной ветке зафиксированы изменения, выполненные текущим днём. То есть только то, что было изменено в коде на момент создания ветки. Безусловно, данная функция весьма привлекательно для работы в проекте. Её выражение способствует небольшим отвлечениям от основной линии не только для экспериментов, но и для возможного последующего анализа.

Изображение № 30 Фиксация текущих изменений в ветке snils



Для того, чтобы в данную ветвь внести все изменения, необходимо их перенести. При этом можно осуществить перенос как в локальную ветвь, так и в удалённый репозиторий GitHub.

Изображение № 31 Перемещение изменений между ветвлениями



Помимо представленных операций, существует возможность слияния веток. Это процедура характерна для командной работы, о которой рассказано в предыдущих главах.

# **Выводы**

Рассматривая современные технологии проектирования программных комплексов, безусловно, необходимо уделить пристальное внимание Git репозиториям и GitHub. Более глубокое изучение подобной технологии способствует не только оптимизации программных продуктов, но и даст возможность в критических ситуациях использовать работоспособные ветви программного продукта.

Периодически могут встречаться ситуации, характерные Agile методологии, где владелец продукта внезапно изменит проект кардинально. Например, проект веб-сайта. В данной ситуации на помощь могут прийти подмодули (submodule) Git. Используя различные подключаемые библиотеки, последнюю весьма сложно модифицировать под свои требования, при этом необходимо подключение библиотек на каждом клиенте. Подмодули, в свою очередь, позволяют содержать репозиторий Git как подкаталог другого репозитория Git. Это позволяет клонировать ещё один репозиторий внутрь проекта.

Ещё одним важным аспектом, при использовании Scrum методологии с использованием GitHub, будет возможность вести суперпроекты. То есть объединять подкаталоги крупного проекта во взаимосвязанные проекты команды. Это позволит более гибко определять отношения между проектами при помощи меток и ветвей суперпроектов.

Безусловно, понимание принципов работы с Git на более низком уровне, упрощает понимание работы Git в целом, и позволяет писать собственные утилиты и сценарии для специфических процессов работы. Git, как контентно-адресуемая файловая система, весьма мощный инструмент, который можно использовать не только как систему управления версиями.

<https://bakunin.com/agile-is/>

https://ru.atlassian.com/agile/scrum

1. Американский программист, директор Lockheed Software Technology Center в Остине, Техас. [↑](#footnote-ref-1)
2. Организация многократной обработки данных, не приводя к вызовам на самих себя. [↑](#footnote-ref-2)
3. Настольный (отдельный компьютер сотрудника) [↑](#footnote-ref-3)
4. Резерв продукта [↑](#footnote-ref-4)
5. Отставание от спринта [↑](#footnote-ref-5)
6. Цель спринта [↑](#footnote-ref-6)
7. Диаграмма сжигания спринта [↑](#footnote-ref-7)
8. Владелец продукта [↑](#footnote-ref-8)
9. Мастер схватки [↑](#footnote-ref-9)
10. Встреча по планированию спринта [↑](#footnote-ref-10)
11. Ежедневная встреча команды [↑](#footnote-ref-11)
12. Обзор спринта [↑](#footnote-ref-12)
13. Относящийся к прошлому (взгляд в пройденные спринты) [↑](#footnote-ref-13)
14. Безопасный хэш-алгоритм [↑](#footnote-ref-14)
15. Основная ветка, в GitHub создаётся по умолчанию [↑](#footnote-ref-15)
16. Опции программы [↑](#footnote-ref-16)
17. URL (Uniform Resource Locator) – единообразный локатор (определитель местонахождения) ресурса [↑](#footnote-ref-17)
18. Hyper Text Transfer Protocol – протокол «передачи гипертекста», протокол прикладного уровня передачи данных по технологии «клиент-сервер». [↑](#footnote-ref-18)
19. Secure Shell («Безопасная оболочка») – протокол прикладного уровня, удалённое управление ОС и туннелирование TCP-соединения. [↑](#footnote-ref-19)
20. Расширение [↑](#footnote-ref-20)