**№1 Основные, вспомогательные и организационные процессы жизненного цикла ПО**

Процессы жизненного цикла по делится на три группы:  
1) Основные:

* Приобретение (действия и задачи заказчика, приобретающего ПО)
* Поставка (действия и задачи поставщика, который снабжает заказчика программным продуктом или услугой)
* Разработка (действия и задачи, выполняемые разработчиком: создание ПО, оформление проектной и эксплуатационной документации, подготовка тестовых и учебных материалов и т. д.)
* Эксплуатация (действия и задачи оператора — организации, эксплуатирующей систему)
* Сопровождение (действия и задачи, выполняемые сопровождающей организацией, то есть службой сопровождения). Сопровождение — внесений изменений в ПО в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.   
  2) Вспомогательные
* Документирование (формализованное описание информации, созданной в течение ЖЦ ПО)
* Управление конфигурацией (применение административных и технических процедур на всем протяжении ЖЦ ПО для определения состояния компонентов ПО, управления его модификациями).
* Обеспечение качества (обеспечение гарантий того, что ИС и процессы ее ЖЦ соответствуют заданным требованиям и утвержденным планам)
* Верификация (определение того, что программные продукты, являющиеся результатами некоторого действия, полностью удовлетворяют требованиям или условиям, обусловленным предшествующими действиями)
* Аттестация (определение полноты соответствия заданных требований и созданной системы их конкретному функциональному назначению)
* Совместная оценка (оценка состояния работ по проекту: контроль планирования и управления ресурсами, персоналом, аппаратурой, инструментальными средствами)
* Аудит (определение соответствия требованиям, планам и условиям договора)
* Разрешение проблем (анализ и решение проблем, независимо от их происхождения или источника, которые обнаружены в ходе разработки, эксплуатации, сопровождения или других процессов)   
  3) Организационные
* Управление (действия и задачи, которые могут выполняться любой стороной, управляющей своими процессами)
* Создание инфраструктуры (выбор и сопровождение технологии, стандартов и инструментальных средств, выбор и установка аппаратных и программных средств, используемых для разработки, эксплуатации или сопровождения ПО)
* Усовершенствование (оценка, измерение, контроль и усовершенствование процессов ЖЦ)
* Обучение (первоначальное обучение и последующее постоянное повышение квалификации персонала)

**№2 Стадии жизненного цикла ПО.**

1. формирование требований к ПО;
2. проектирование (разработка системного проекта);
3. реализация (может быть разбита на подэтапы: детальное проектирование, кодирование);
4. тестирование (может быть разбито на автономное и комплексное тестирование и интеграцию);
5. ввод в действие (внедрение)
6. эксплуатация и сопровождение;
7. снятие с эксплуатации;

**№3. Базовые стратегии разработки ПО, выбор модели.**

Начальный этап: кодирование – устранение ошибок

Три базовые стратегии: каскадная, инкрементная, эволюционная. Определяются характеристиками: проекта, требований к продукту, команды разработчиков, команды пользователей.

**№4. Каскадная стратегия разработки: достоинства и недостатки**

Основными достоинствами каскадной стратегии, проявляемыми при разработке соответствующего ей проекта, являются:

1. стабильность требований в течение ЖЦ разработки;
2. необходимость только одного прохода этапов разработки, что обеспечивает простоту применения стратегии;
3. простота планирования, контроля и управления проектом;
4. доступность для понимания заказчиками.

К основным недостаткам каскадной стратегии, проявляемым при ее использовании в проекте, ей не соответствующем, следует отнести:

1. сложность полного формулирования требований в начале процесса разработки и невозможность их динамического изменения на протяжении ЖЦ;
2. линейность структуры процесса разработки; разрабатываемые ПС или системы обычно слишком велики и сложны, чтобы все работы по их созданию выполнять однократно; в результате возврат к предыдущим шагам для решения возникающих проблем приводит к увеличению финансовых затрат и нарушению графика работ;
3. непригодность промежуточных продуктов для использования;
4. недостаточное участие пользователя в процессе разработки ПС – только в самом начале (при разработке требований) и в конце (во время приемочных испытаний); это приводит к невозможности предварительной оценки пользователем качества программного средства или системы.

Использование данной стратегии наиболее эффективно в следующих случаях:

1. при разработке проектов с четкими, неизменяемыми в течение ЖЦ требованиями и понятной реализацией;
2. при разработке проектов невысокой сложности, например: создание программного средства или системы такого же типа, как уже разрабатывались разработчиками; создание новой версии уже существующего программного средства или системы; перенос уже существующего продукта на новую платформу;
3. при выполнении больших проектов в качестве составной части моделей ЖЦ, реализующих другие стратегии разработки.

**№5 Инкрементная модель.**

Данная стратегия основана на полном определении всех требований к разрабатываемому программному средству (системе) в начале процесса разработки. Однако полный набор требований реализуется постепенно в соответствии с планом в последовательных циклах разработки. Результат каждого цикла называется инкрементом.

Достоинства:

1. возможность получения функционального продукта после реализации каждого инкремента;
2. короткая продолжительность создания инкремента; это приводит к сокращению сроков начальной поставки, позволяет снизить затраты на первоначальную и последующие поставки программного продукта;
3. предотвращение реализации громоздких спецификаций требований; стабильность требований во время создания определенного инкремента; возможность учета изменившихся требований;
4. снижение рисков по сравнению с каскадной стратегией;
5. Включение в процесс пользователей, что позволяет оценить функциональные возможности продукта на более ранних этапах разработки и в конечном итоге приводит к повышению качества программного продукта, снижению затрат и времени на его разработку.

Недостатки :

1. необходимость полного функционального определения системы или программного средства в начале ЖЦ для обеспечения планирования инкрементов и управления проектом;
2. возможность текущего изменения требований к системе или программному средству, которые уже реализованы в предыдущих инкрементах;
3. сложность планирования и распределения работ;
4. проявление человеческого фактора, связанного с тенденцией к оттягиванию решения трудных проблем на поздние инкременты, что может нарушить график работ или снизить качество программного продукта;

Область применения :

1. при разработке проектов, в которых большинство требований можно сформулировать заранее, но часть из них могут быть уточнены через определенный период времени.
2. при разработке сложных проектов с заранее сформулированными требованиями; для них разработка системы или программного средства за один цикл связана с большими трудностями;
3. при необходимости быстро поставить на рынок продукт, имеющий базовые функциональные свойства;
4. при разработке проектов с низкой или средней степенью рисков;
5. при выполнении проекта с применением новых технологий.

**6.Эволюционная стратегия.**

Эволюционная стратегия представляет собой многократный проход этапов разработки. Данная стратегия основана на частичном определении требований к разрабатываемому программному средству или системе в начале процесса разработки. Требования постепенно уточняются в последовательных циклах разработки. Результат каждого цикла разработки обычно представляет собой очередную поставляемую версию программного средства или системы.

Достоинства:

1. возможность уточнения и внесения новых требований в процессе разработки;
2. пригодность промежуточного продукта для использования;
3. возможность управления рисками;
4. обеспечение широкого участия пользователя в проекте, начиная с ранних этапов, что минимизирует возможность разногласий между заказчиками и разработчиками и обеспечивает создание продукта высокого качества;
5. реализация преимуществ каскадной и инкрементной стратегий.

Недостатки:

1. неизвестность точного количества необходимых итераций и сложность определения критериев для продолжения процесса разработки на следующей итерации; это может вызвать задержку реализации конечной версии системы или программного средства;
2. сложность планирования и управления проектом;
3. необходимость активного участия пользователей в проекте, что реально невсегда осуществимо;
4. необходимость в мощных инструментальных средствах и методах прототипирования;
5. возможность отодвигания решения трудных проблем на последующие циклы, что может привести к несоответствию полученных продуктов требованиям заказчиков.

Область применения:

* При разработке проектов, для которых требования слишком сложны, неизвестны заранее, непостоянны или требуют уточнения;
* при разработке сложных проектов, в том числе: больших долгосрочных проектов; проектов по созданию новых, не имеющих аналогов ПС или систем;
* проектов со средней и высокой степенью рисков;
* проектов, для которых нужна проверка концепции, демонстрация технической осуществимости или промежуточных продуктов;
* при разработке проектов, использующих новые технологии.

**7. Гибкие технологии разработки ПО: Agile**

Гибкая методология разработки (англ. Agile software development, agileметоды) — серия подходов к разработке программного обеспечения, ориентированных на использование итеративной разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля. Существует несколько методик, относящихся к классу гибких методологий разработки, в частности экстремальное программирование, DSDM, Scrum, FDD.

**8. XP тестирование и рефакторинг**

Тестирование (testing). В отличие от большинства остальных методологий тестирование в XP — одно из важнейших составляющих. Экстремальный подход заключается в том, что тесты пишутся до написания кода. Каждый модуль обязан иметь unit test — тест данного модуля; таким образом, в XP осуществляется regression testing (возвратное тестирование, «неухудшение качества» при добавлении функциональности). Большинство ошибок исправляются на стадии кодирования. Еще один важный принцип: тест определяет код, а не наоборот (такой подход носит название test-driven development), то есть кусок кода кладется в хранилище тогда и только тогда, когда все тесты прошли успешно, в противном случае данное изменение кода отвергается.

Рефакторинг (refactoring). Рефакторинг — это оптимизация существующего кода в сторону упрощения, что предусматривает постоянную работу по упрощению кода. Сохраняя код прозрачным и определяя его элементы всего один раз, программисты сокращают число ошибок, которые впоследствии придется устранять. При реализации каждого нового свойства системы программист должен подумать над тем, можно ли упростить существующий код и как это поможет реализовать новое свойство. Кроме того, нельзя совмещать рефакторинг с дизайном: если создается новый код, рефакторинг надо отложить

**9. ХР code style**

Стандарты кодирования (coding conventions). Стандарты кодирования нужны для обеспечения других практик: коллективного владения кодом, парного программирования и рефакторинга. Без единого стандарта выполнять эти практики как минимум сложнее, а в реальности вообще невозможно: группа будет работать в режиме постоянной нехватки времени. Детальные стандарты не требуются, необходимо стандартизировать только важные вещи. Определение наиболее важных объектов стандартизации в XP субъективно.

**10. Экстремальное программирование (XP): парное программирование и continuous integration**

Парное программирование (pair programming) — одна из самых известных XPпрактик. Все программисты должны работать в парах: один пишет код, другой смотрит. Таким образом, необходимо размещать группу программистов в одном месте, что легче всего сделать на территории заказчика (все необходимые члены команды географически находятся в одном месте); XP наиболее успешно работает в нераспределенных коллективах программистов и пользователей.

Непрерывная интеграция (continuous integration). Интеграция новых частей системы должна происходить как можно чаще, как минимум раз в несколько часов. Основное правило интеграции следующее: интеграцию можно производить, если все тесты проходят успешно. Если тесты не проходят, то программист должен либо внести исправления и тогда интегрировать составные части системы, либо вообще не интегрировать их. Правило это жесткое и однозначное — если в созданной части системы имеется хотя бы одна ошибка, то интеграцию производить нельзя.

***Парное программирование*** (pair programming) — одна из самых известных XPпрактик. Все программисты должны работать в парах: один пишет код, другой смотрит. Таким образом, необходимо размещать группу программистов в одном месте, что легче всего сделать на территории заказчика (все необходимые члены команды географически находятся в одном месте); XP наиболее успешно работает в нераспределенных коллективах программистов и пользователей.

Интенсивная разработка малыми группами (не больше 10 человек) и ***парное программирование*** (когда два программиста вместе создают код на одном общем рабочем месте), активное общение в группе и между группами. Все это нацелено на как можно более раннее обнаружение проблем (как ошибок, так и срыва сроков). Парное программирование направлено на решение задачи стабилизации проекта. При применении XP методологии высок риск потери кода по причине ухода программиста, не выдержавшего интенсивного графика работы. В этом случае второй программист из пары играет роль «наследника» кода. Немаловажно и то, как именно распределены группы в рабочем пространстве – в XP используется открытое рабочее пространство, которое предполагает быстрый и свободный доступ всех ко всем.  Обратная связь с заказчиком, представитель которого фактически

***Continuous integration***

***Непрерывная интеграция*** (CI, англ. Continuous Integration) — это практика разработки программного обеспечения, которая заключается в слиянии рабочих копий в общую основную ветвь разработки несколько раз в день и выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

**11.Экстремальное программирование (XP): соглашение о кодировании и коллективное владение кодом**

***Коллективное владение кодом*** (collective code ownership). Каждый программист в коллективе XP должен иметь доступ к коду любой части системы и вносить изменения в любой код. Обязательное правило: если программист внес изменения и система после этого работает некорректно, то именно этот программист должен исправить ошибки. В противном случае работа системы уподобится тотальному хаосу.

***Соглашение о кодировании***

Вы в команде, которая работает над данным проектом продолжительное время. Люди приходят и уходят. Никто не кодирует в одиночку и код принадлежит всем. Всегда будут моменты, когда необходимо будет понять и скорректировать чужой код. Разработчики будут удалять или изменять дублирующий код, анализировать и улучшать чужие классы и т.п. Со временем нельзя будет сказать кто автор конкретного класса.

Следовательно, все должны подчиняться общим стандартам кодирования - форматирование кода, именование классов, переменных, констант, стиль комментариев. Таким образом, мы будем уверены, что внося изменения в чужой код (что необходимо для агрессивного и экстремального продвижения вперед) мы не превратим его в Вавилонское Столпотворение.

Вышесказанное означает, что все члены команды должны договориться о общих стандартах кодирования. Неважно каких. Правило заключается в том, что все им подчиняются. Те кто не желает их соблюдать покидает команду.

**12. Scrum: основные характеристики**

***Основные характеристики***

* Самоорганизующиеся команды
* Продукт разрабатывается серией “спринтов” , каждый не больше месяца
* Все требования записываются в виде единого списка “бэклога продукта”
* Инженерные практики не являются частью Скрам
* Использует простые правила для создания гибкой среды разработки проектов
* Один из “Agile процессов”

***13 .Структура Скрам Роли***

**1.Владелец продукта- Один человек**

* Определяет требования к продукту
* Определяет дату релиза и наполненность • Ответственен за доходность проекта (ROI)
* Приоритезирует требования, исходя из их рыночной ценности
* Корректирует приоритеты на каждой итерации, если необходимо
* Принимает работу

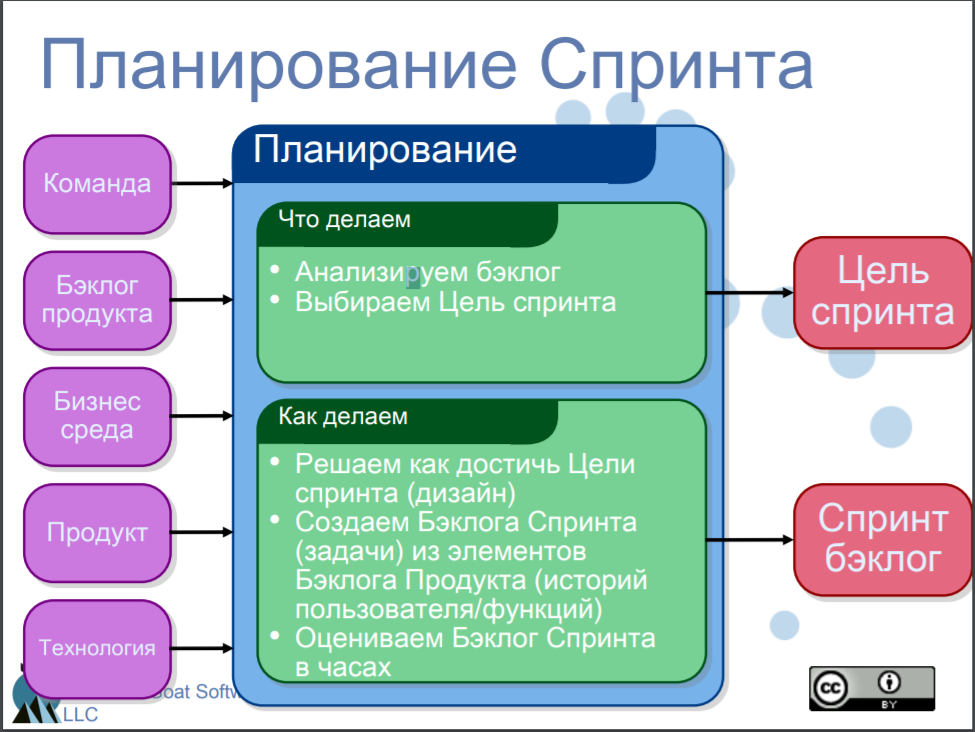
2.Скрам-мастер - Представляет руководство проекта

* Ответственен за внедрение ценностей и практик Скрам
* Не раздает задания
* Устраняет препятствия
* Ответственен за эффективность работы команды
* Обеспечивает видимость и прозрачность ситуации в команде
* Защищает команду от внешних воздействий

3.Команда - Обычно 5-9 человек

* Кросс функциональная
* программисты, тестировщики, дизайнеры...
* Заняты полный рабочий день
* Могут быть исключения (например, администратор базы данных) • Команды самоорганизуюются
* В идеале, нет специальных ролей
* Состав команды может меняться только между спринтами

14.***Структура Скрам Ритуалы***

1.Планирование спринта 

2.Обзор спринта -Команда представляет, что было сделано за спринт

* Фокус на результат, а не процесс
* Обычно принимает форму демонстрации
* Неформально
* Максимум 2 часа на подготовку
* Без слайдов
* Вся команда участвует
* Приглашены все, кому может быть интересно

3.Спринт ретроспектива -Периодический пересмотр того, что работает, а что нет

* Обычно 15-30 минут
* После каждого спринта
* Вся команда участвует
* Возможно, приглашен Владелец продукта, заказчики или кто-то из менеджмента компании

4.Ежедневный Скрам -Характеристики

* Ежедневно
* 15 минут
* Стоя
* Не для решения проблем
* Приглашены все желающие
* Только участники команды могут говорить (владелец продукта - тоже часть команды)
* ScrumMaster лишь ведет собрание

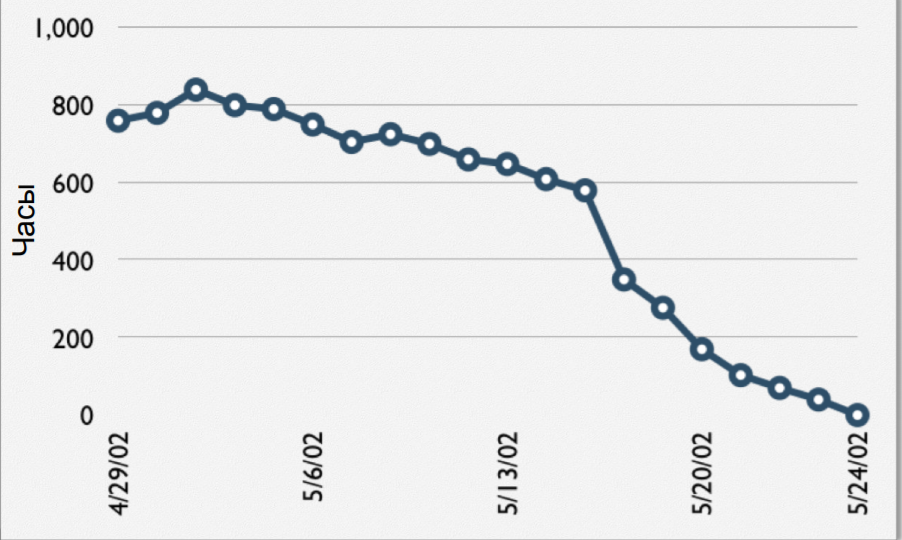
15.***Структура Скрам Артефакты*** 1.Бэклог продукта- Требования

* Список желательной функциональности
* В идеале написан так, что каждый элемент имеет значение для конечного пользователя
* Приоритеты выставляются Владельцем продукта
* Приоритеты обновляются в начале спринта  2.Спринт бэклог -Цель спринта

Короткое предложение, описывающее, на чем будет сфокусирована работа во время спринта….

Управление бэклогом спринта:

* Члены команды выбирают работу на свой выбор
* Задачи никогда не назначаются принудительно
* Оценка оставшейся работы ежедневно обновляется
* Любой член команды может добавить, удалить или изменить элементы Бэклога Спринта
* Задачи на спринт могут появляться в ходе работы
* Если задача не понятна, то этому элементу бэклога резервируется больше времени и он разбивается на составные части позже
* Оценка оставшейся работы обновляется, по мере того как узнаем больше о задачах  **3. Burndown charts**



16.**Scrum: масштабируемость**

Обычно команда состоит из 7 ± 2 человек

Масштабируемость за счет групп команд

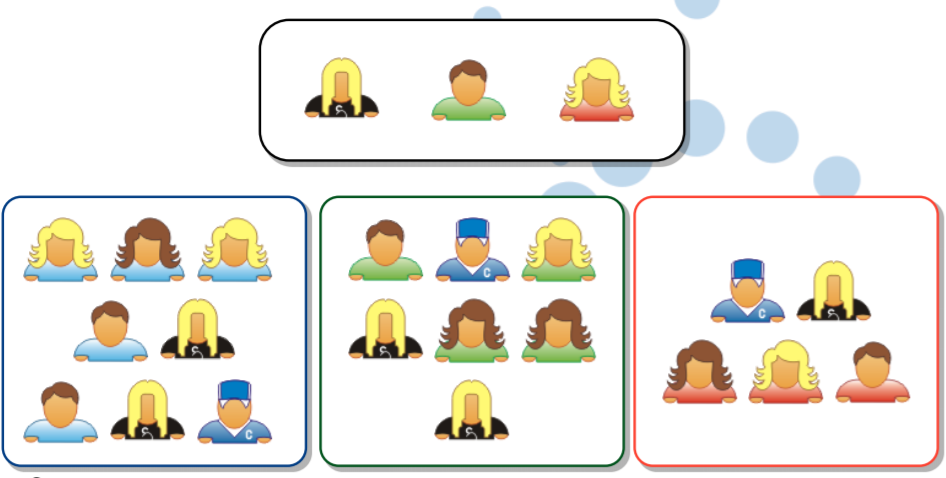
Факторы расширения

Тип приложения

Размер команды

Рассредоточенность команды

Продолжительность проекта

Scrum использовался в проектах со сложной структурой, где работало более 500 человек 

**17. Тестирование ПО: разработка теста для функции по ее спецификации**

Спецификация — это текстовый файл с описанием того, что нужно протестировать в тестовых данных. В ней указывается какие результаты должна получить программа. Тестовый код находит реальные, вычисленные на живом коде результаты. А тестовый движок производит сверку спецификации и вычисленных результатов.

**18. Тестирование ПО: статическое, динамическое тестирование; классификация по степени важности**

**1. Статическое** **тестирование**– тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться. При этом само тестирование может быть как ручным, так и автоматизированным.

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации. Для этого типа тестирования в некоторых случаях даже не нужен компьютер – например, при проверке требований.

Большинство статических техник могут быть использованы для «тестирования» любых форм документации, включая вычитку кода, инспекцию проектной документации, функциональной спецификации и требований.

Даже статическое тестирование может быть автоматизировано – например, можно использовать автоматические средства проверки синтаксиса программного кода.

*Виды* статического тестирования:

– вычитка исходного кода программы;

– проверка требований.

**2. Динамическое тестирование** – тип тестирования, который предполагает запуск программного кода. Таким образом, анализируется *поведение* программы во время ее работы.

Для выполнения динамического тестирования необходимо чтобы тестируемый программный код был *написан*,*скомпилирован*и*запущен*. При этом, может выполняться проверка внешних параметров работы программы: загрузка процессора, использование памяти, время отклика и т.д. – то есть, ее производительность.

Динамическое тестирование является частью процесса валидации программного обеспечения.

Кроме того динамическое тестирование может включать разные подвиды, каждый из которых зависит от:

• Доступа к коду (тестирование черным, белым и серым ящиками).

• Уровня тестирования (модульное интеграционное, системное, и  приемочное тестирование).

• Сферы использования приложения (функциональное, нагрузочное, тестирование

безопасности и пр.).

**19. Тестирование ПО: методы белого и черного ящика**

***Black box*** Стратегия, основанная на требованиях и спецификациях. Не требует знания реализации. 

***White box*** Стратегия, основанная на знании реализации тестируемого продукта. Требует навыков программирования.

**20. Системы контроля версий: история развития (RCS, CVS, SVN и git), локальные, централизованные и распределенные системы**

**1) Локальные системы контроля версий**

**RCS (Revision Control System, Система контроля ревизий)** была разработана в начале 1980-х годов Вальтером Тичи (Walter F. Tichy). Система позволяет хранить версии только одного файла, таким образом управлять несколькими файлами приходится вручную.

ci (от check-in, регистрировать):

**$ ci file.txt co**

(от check-out)

**$ co file.txt**

Недостатки:

* Работа только с одним файлом, каждый файл должен контролироваться отдельно;
* Неудобный механизм одновременной работы нескольких пользователей с системой

**2) Централизованные системы контроля версий: CVS**

**CVS (Concurrent Versions System, Система совместных версий)** Дик Грун (Dick Grune) разработал CVS в середине 1980-х. Для хранения индивидуальных файлов CVS (также как и RCS) использует файлы в RCS формате, но позволяет управлять группами файлов расположенных в директориях. Также CVS использует клиент-сервер архитектуру в которой вся информация о версиях хранится на сервере.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ (система контроля версий). Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми он работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы — когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

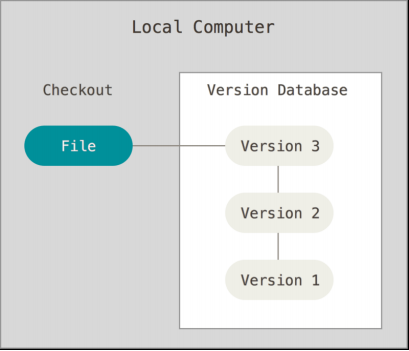
**$ cvs checkout path-in-repository со = checkout**

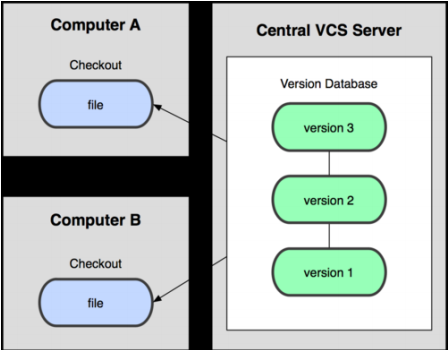
**$ cvs commit -m "Some changes“ ci = commit**

**$ cvs update**

Недостатки:

* Так как версии хранятся в файлах RCS нет возможности сохранять версии директорий.
* Перемещение, или переименование файлов не подвержено контролю версий.





**SVN (Subversion)** свободная централизованная система управления версиями, официально выпущенная в 2004 году компанией CollabNet. Из наиболее значительных изменений по сравнению с CVS можно отметить:

* Атомарное внесение изменений (commit). В случае если обработка коммита была прервана не будет внесено никаких изменений.
* Переименование, копирование и перемещение файлов сохраняет всю историю изменений.
* Директории, символические ссылки и мета-данные подвержены контролю версий.
* Эффективное хранение изменений для бинарных файлов.

Команды SVN во многом похожи на CVS:

**$ svn checkout path-in-repository**

**$ svn commit -m "Some changes“**

**$ svn up**

**3) Распределенные системы контроля версий: git (2005)**

Клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени): они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Основные требования к новой системе были следующими:

* Скорость
* Простота дизайна
* Поддержка нелинейной разработки (тысячи параллельных веток)
* Полная распределённость
* Возможность эффективной работы с такими большими проектами, как ядро Linux (как по скорости, так и по размеру данных)

***Помощь:***

**$ git help**

**$ git --help**

**$ man git-**

**21. Git: создание репозитория + фиксация изменений, стадии процесса**

***Создание репозитория:***

**$ git init** Эта команда создаёт в текущей директории новую поддиректорию с именем .git

Эта команда создаёт в текущей директории новую поддиректорию с именем .git, содержащую все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем.

***Добавление файлов под версионный контроль:***

**$ git add**

**$ git commit -m 'initial project version'**

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду git clone. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется "clone", а не "checkout". Это важное различие – вместо того, чтобы просто получить рабочую копию, Git получает копию практически всех данных, которые есть на сервере. При выполнении git clone с сервера забирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования

git clone [url]



Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда git status.

$ git status

On branch master

nothing to commit, working directory clean

Это означает, что у вас чистый рабочий каталог, другими словами – в нем нет отслеживаемых измененных файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере.

$ git add README

Если вы снова выполните команду status, то увидите, что файл README теперь отслеживаемый и индексированный:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл ``CONTRIBUTING.md`` и после этого снова выполните команду git status, то результат будет примерно следующим:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

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: CONTRIBUTING.md

Файл ``CONTRIBUTING.md`` находится в секции ``Changes not staged for commit`` — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду git add. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния.

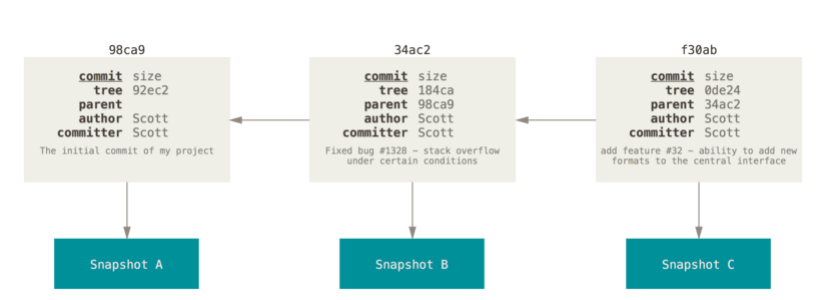
Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т.п.). В таком случае, вы можете создать файл .gitignore. с перечислением шаблонов соответствующих таким файлам.

К шаблонам в файле .gitignore применяются следующие правила:

* Пустые строки, а также строки, начинающиеся с #, игнорируются.
* Можно использовать стандартные glob шаблоны.
* Можно начать шаблон символом слэша (/) чтобы избежать рекурсии.
* Можно заканчивать шаблон символом слэша (/) для указания каталога.
* Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Если результат работы команды git status недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff. Позже мы рассмотрим команду git diff подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь фиксировать. Если git status отвечает на эти вопросы в самом общем виде, перечисляя имена файлов, git diff показывает вам непосредственно добавленные и удалённые строки — собственно заплатку (patch). Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить git diff --staged.

**22. Git: создание ветки (branch)**

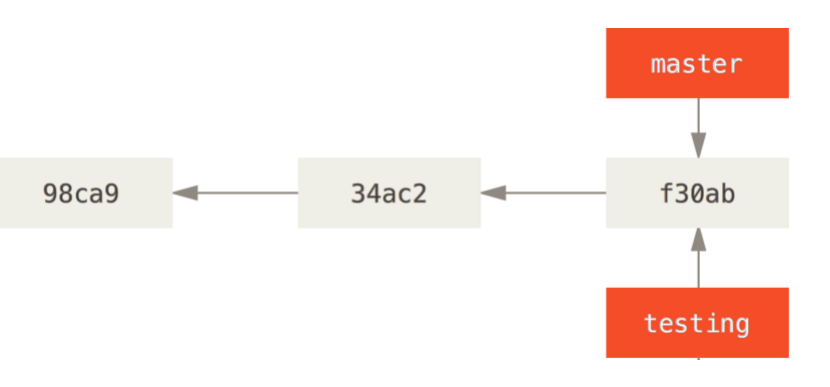


* Ветка (branch) в Git — это легко перемещаемый указатель на один из этих коммитов. Имя основной ветки по умолчанию в Git — master
* Когда вы делаете коммиты, то получаете основную ветку, указывающую на ваш последний коммит. Каждый коммит автоматически двигает этот указатель вперед.

**$ git branch testing**

**Или**

**$ git checkout –b testing**



Команда git branch только создает новую ветку. Переключения не происходит.

В Git HEAD - это указатель на локальную ветку, в которой вы находитесь.

Указатель на вашу ветку “testing” переместился вперед, а “master” все еще указывает на тот коммит, где вы были в момент выполнения команды git checkout для переключения веток.

$ git checkout master

Эта команда сделала две вещи. Она переместила указатель HEAD назад на ветку “master” и вернула файлы в рабочем каталоге в то состояние, которое было сохранено в снимке (snapshot), на который указывает ветка. Это также означает, что все изменения, вносимые с этого момента, будут отнесены к старой версии проекта. Другими словами, откатилась вся работа, выполненная в ветке “testing”, а вы можете продолжать в другом направлении.

**$ git branch -d hotfix – удаление ветки hotfix**

Команда git branch делает несколько больше, чем просто создаёт и удаляет ветки. При запуске без параметров, вы получите простой список имеющихся у вас веток:

**$ git branch**

**iss53**

**\* master**

**testing**

**$ git branch –v**

**iss53 93b412c fix javascript issue**

**\* master 7a98805 Merge branch 'iss53'**

**testing 782fd34 add scott to the author list in the readmes**

Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду git branch --merged:

**$ git branch --merged**

**iss53**

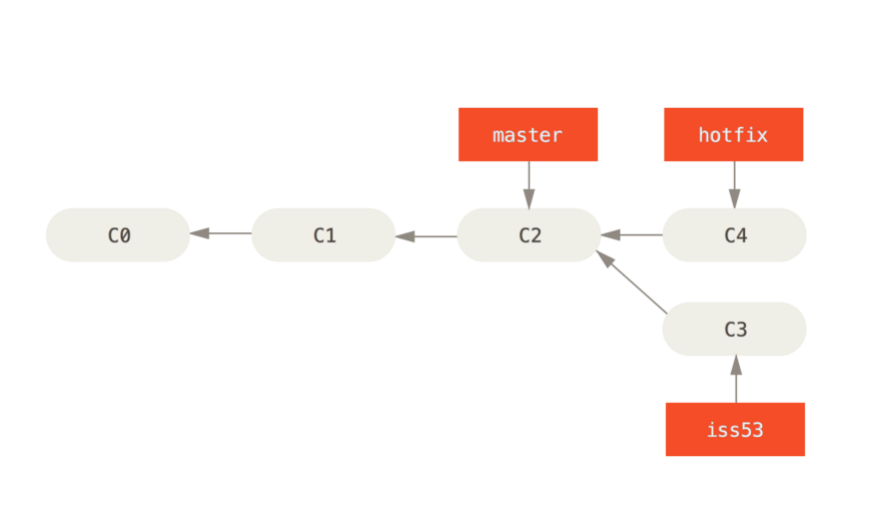
**\* master**

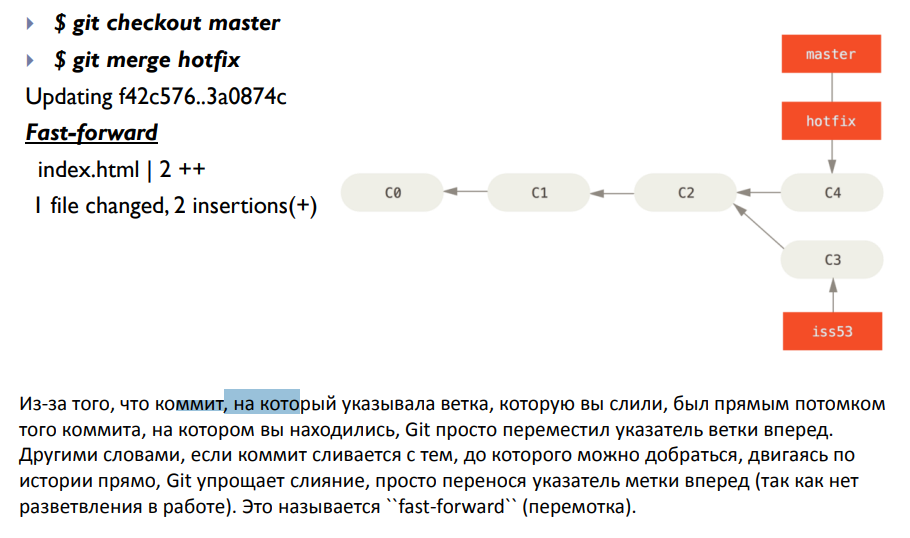
Те ветки из этого списка, перед которыми нет символа \*, можно смело удалять командой git branch –d Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, выполните команду git branch --no-merged:

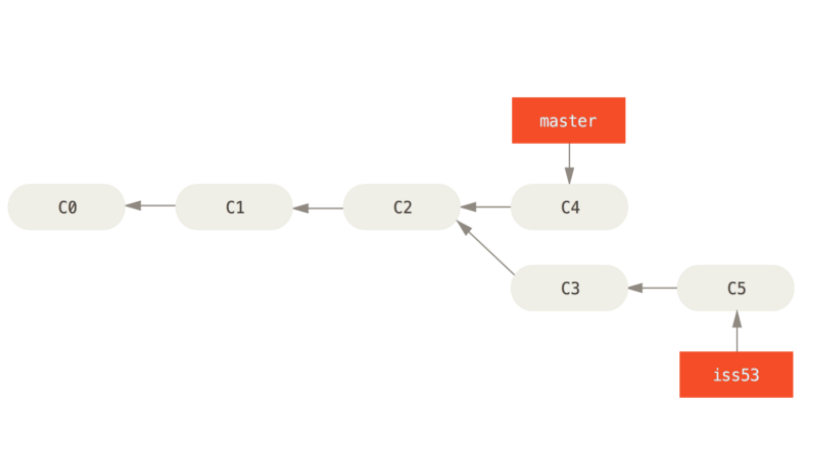
**$ git branch --no-merged**

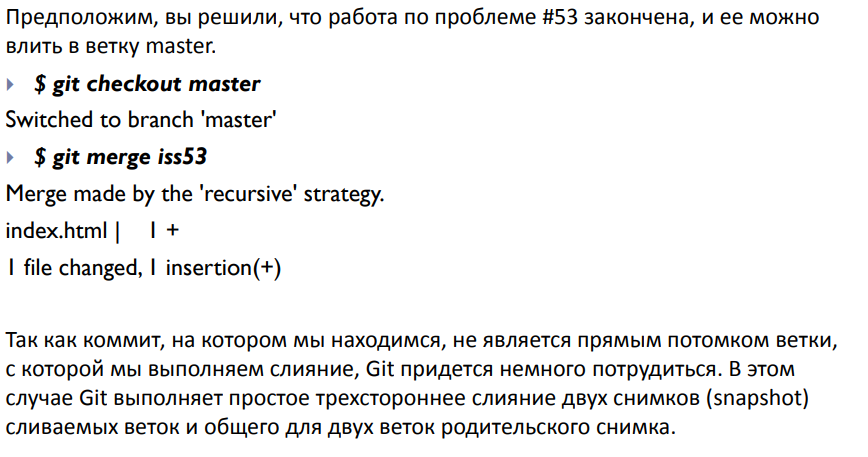
**testing**

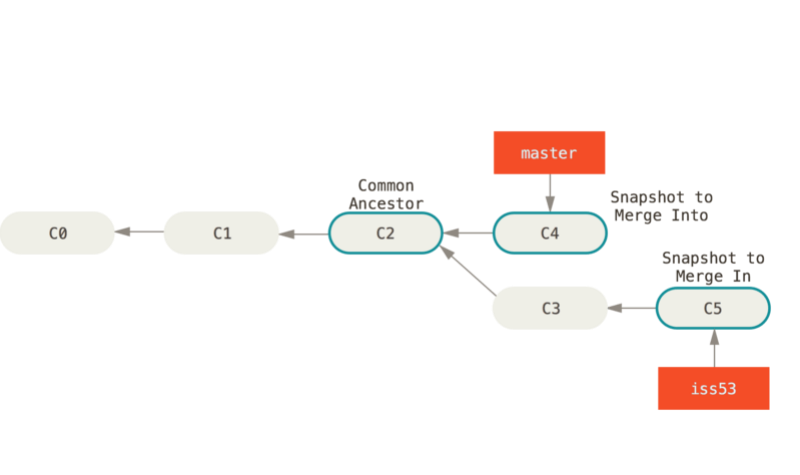
**23. Git: объединение веток (merge)**

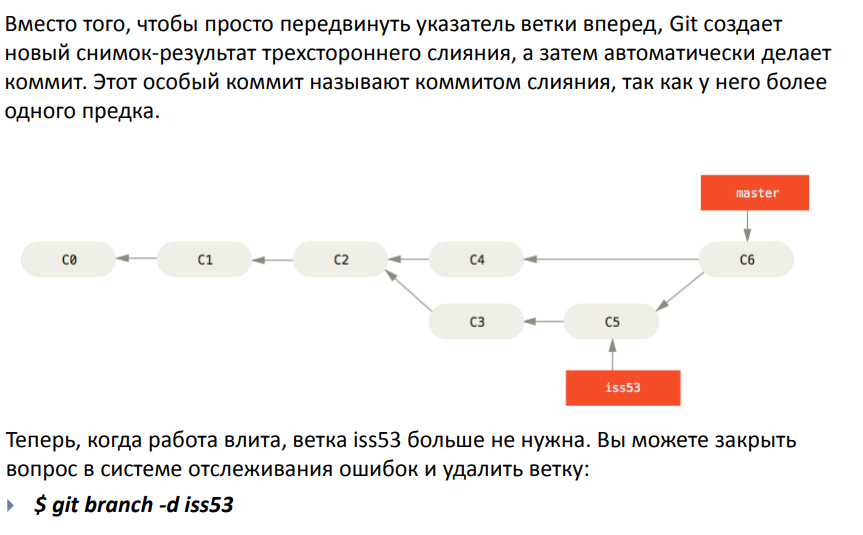




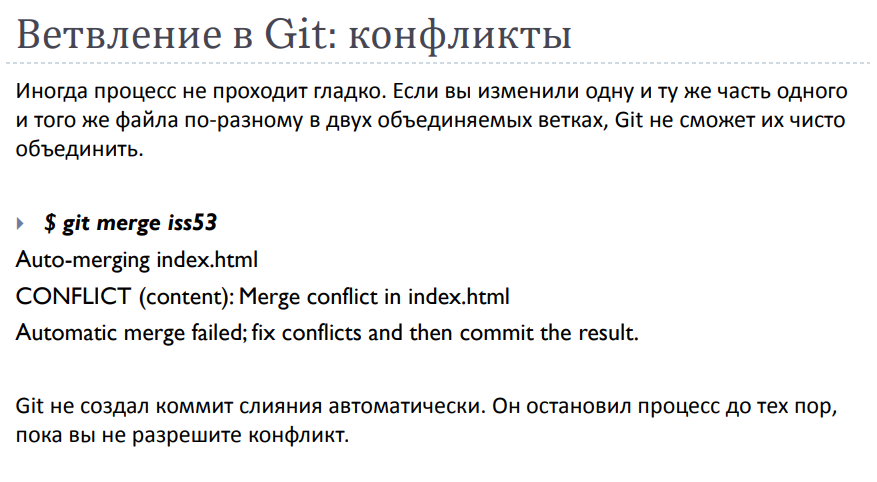


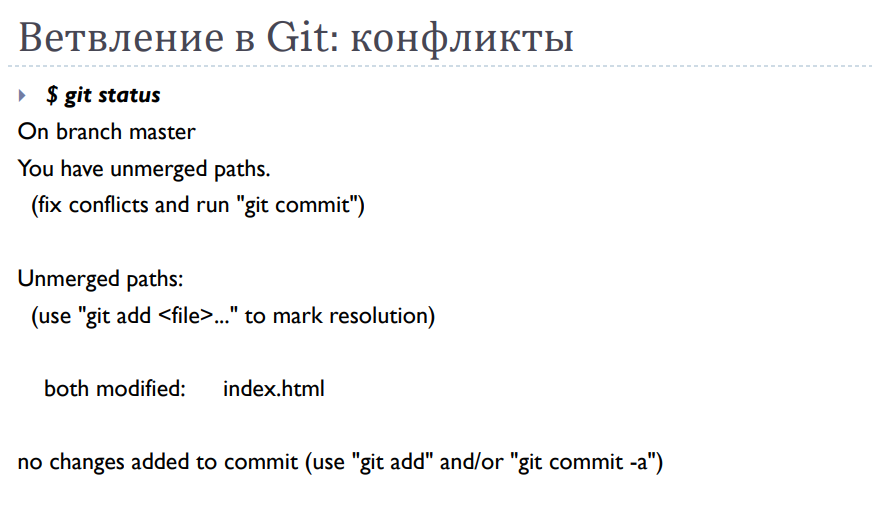






**24. Git: конфликты и способы их разрешения**







Если это вас устраивает, и вы убедились, что все, где были конфликты, подготовлено (staged), можете ввести git commit, чтобы завершить коммит слияния. Комментарий к коммиту по умолчанию выглядит примерно так:

**25. Git: работа с удаленным репозиторием (git fetch & git pull)**

Удалённые ветки — это ссылки (pointers) на состояние веток в ваших удалённых репозиториях. Это локальные ветки, которые нельзя перемещать; они двигаются автоматически всякий раз, когда вы осуществляете связь по сети. Удалённые ветки действуют как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним. Они выглядят как (имя удал. репоз.)/(ветка).

Например, если вы хотите посмотреть, как выглядела ветка master на сервере origin во время последнего соединения с ним, проверьте ветку origin/master.

**Git fetch**

Для получения данных из удалённых проектов, следует выполнить:

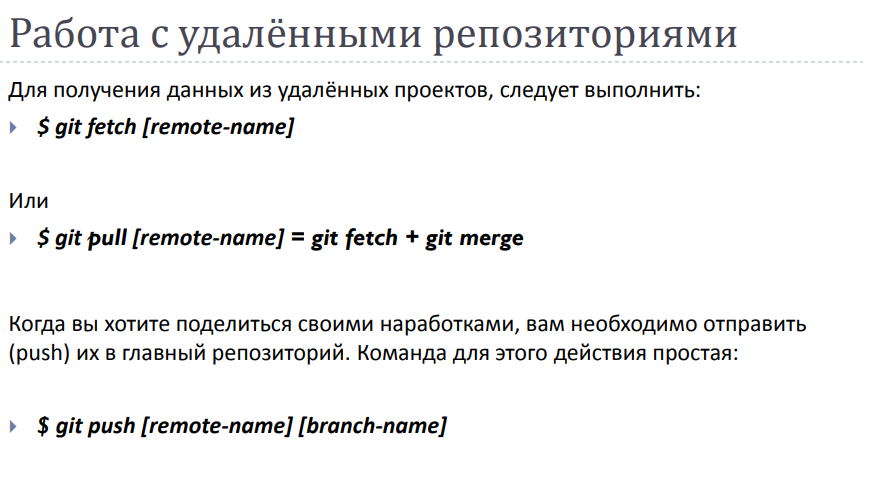
**$ git fetch [имя удал. сервера]**

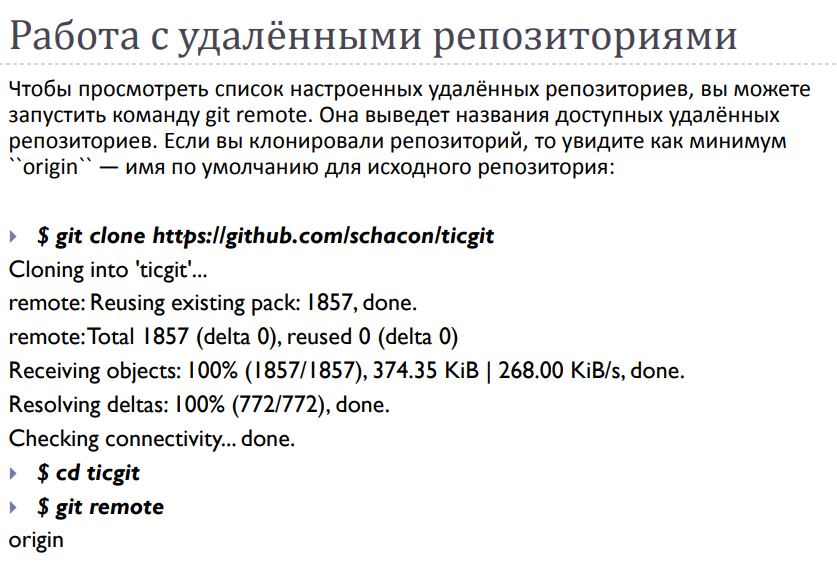
Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты.

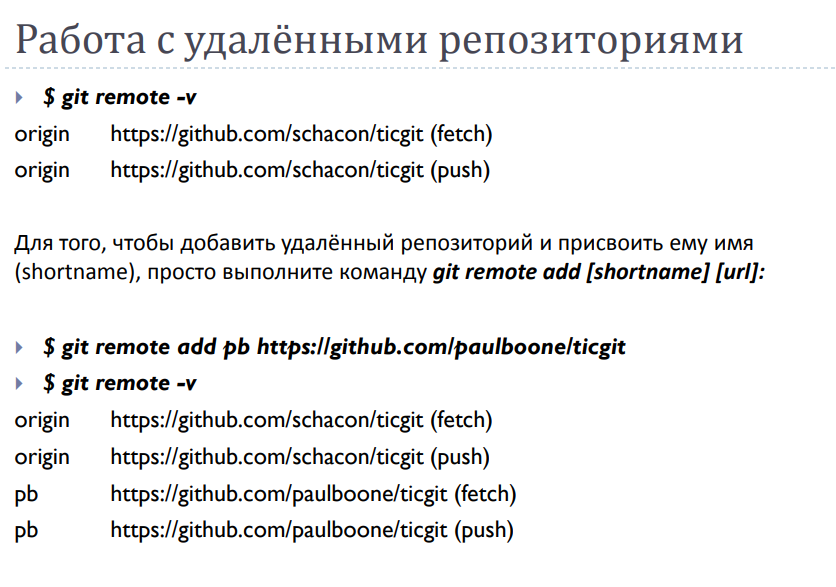
Когда вы клонируете репозиторий, команда clone автоматически добавляет этот удалённый репозиторий под именем origin. Таким образом, git fetch origin извлекает все наработки, отправленные (push) на этот сервер после того, как вы склонировали его (или получили изменения с помощью fetch). Важно отметить, что команда fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы, с помощью git merge.

**Git pull**

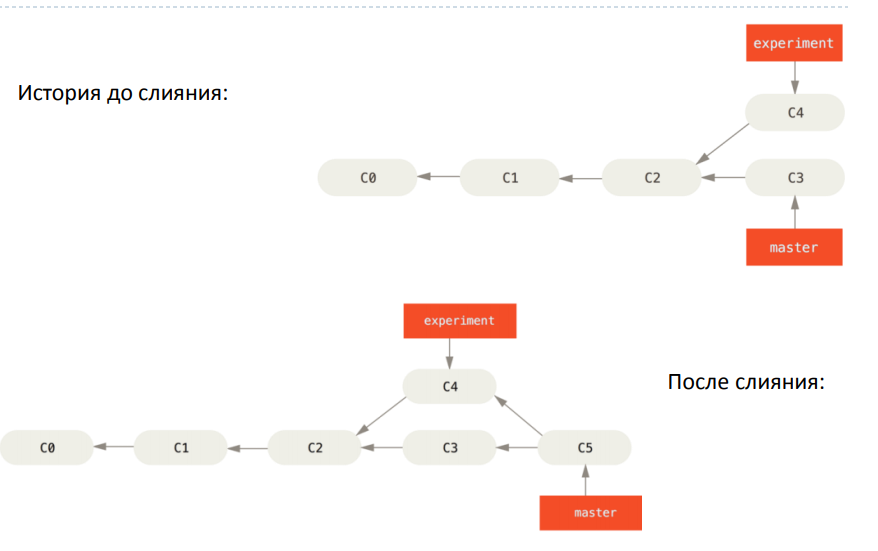
Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду git pull. Она автоматически извлекает и затем сливает данные из удалённой ветки в вашу текущую ветку. Этот способ может для вас оказаться более простым или более удобным. К тому же по умолчанию команда git clone автоматически настраивает вашу локальную ветку master на отслеживание удалённой ветки master на сервере, с которого вы клонировали (подразумевается, что на удалённом сервере есть ветка master). Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, **и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете**.

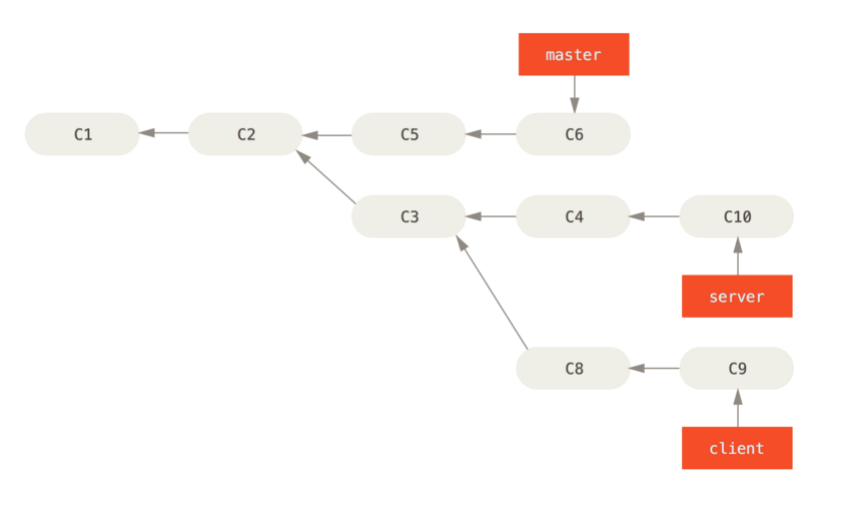
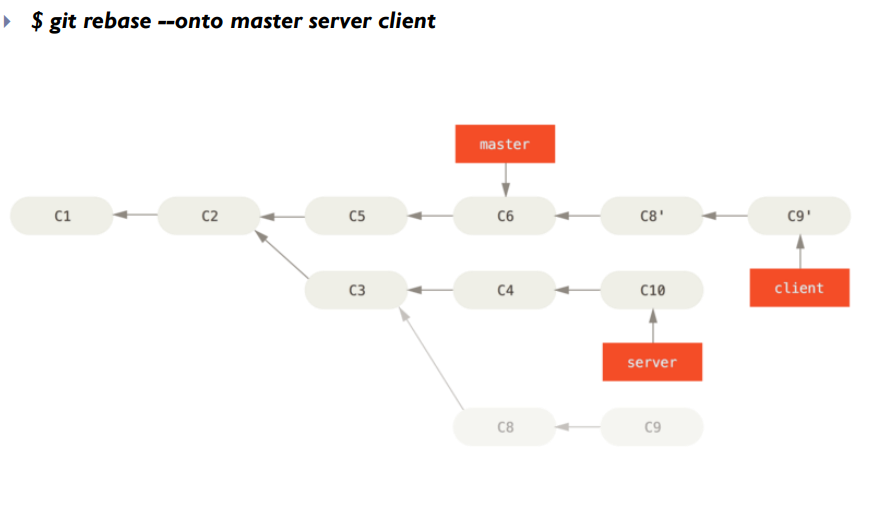






**26. Git: работа с удаленным репозиторием (git rebase)**

**27.  Работа с удаленным репозиторием (git push)**



**28.** Git: [модель ветвления](http://nvie.com/posts/a-successful-git-branching-model/)

В основе модели лежит две ветки:

Master

Develop



origin / master является основной ветвью, где исходный код HEAD всегда отражает состояние готовности к выпуску.

origin/develop является основной отраслью, где исходный код HEAD всегда отражает состояние с последними внесенными изменениями разработки.

Как только исходный код в ветке develop достигает стабильной точки и готов к выпуску, все изменения должны быть слиты обратно в master , а затем помечены номером выпуска.

Параллельно с основными ветками существуют вспомогательные ветки для поддержания параллельной разработки между членами команды такие, как features, prepare for release, hotfix.

**Feature branches**

Может исходить из develop, а потом сливаться обратно в develop. Используются для разработки новых функций для предстоящей или отдаленной будущей версии. Суть ветви заключается в том, что она существует до тех пор, пока функция находится в разработке, но в конечном итоге будет объединена в develop или удалена (в случае провальной функции).



### Release branches

Может исходить из:

develop

Сливаться в:

develop и  master

Предназначена для соединения всех функций и их исправление, если это необходимо.

### Hotfix branches

Может исходить из:

master

Сливаться в:

develop and master

Эта ветка очень схожа с веткой release, но создается в случае, если необходимо срочно внести изменение в текущую релизную версию (исправление ошибок и тд)



**29. Системы автоматизации сборки: цели, задачи, примеры**

Ручная сборка проекта и формирование исполнимого образа для большого проекта (содержащего тысячи исходных файлов) являются крайне рутинной работой. При этом высока вероятность, что на этом этапе будут внесены ошибки. То же касается и вопроса многократного повторения однотипных действий по формированию исполнимого образа приложения – при выполнении в ручном режиме человек может забыть выполнить то или иное действие (скопировать файл, запустить компилятор, обновить ресурсы и т.п.), что потенциально может привести к передаче в тест несогласованной сборки (содержащей различные (иногда несовместимые друг с другом) бинарные версии исполнимых файлов, конфликтующие конфигурации, устаревшие ресурсы и т.п.). При этом подобные нестыковки способны привести к плавающим и трудно заметным дефектам, проявляющимся во время выполнения программы. В лучшем случае эти дефекты будут обнаружены в момент тестирования, в худшем – могут проявиться на этапе эксплуатации. Таким образом, ручная сборка современных программных комплексов является неэффективной ввиду значительного объема рисков, вызванных человеческим фактором (например, усталостью).

Ручная сборка программы приводит к удлинению цикла Edit - Compile – Test, что усугубляется в проектах с большим объемом кода, использующим шаблонные библиотеки

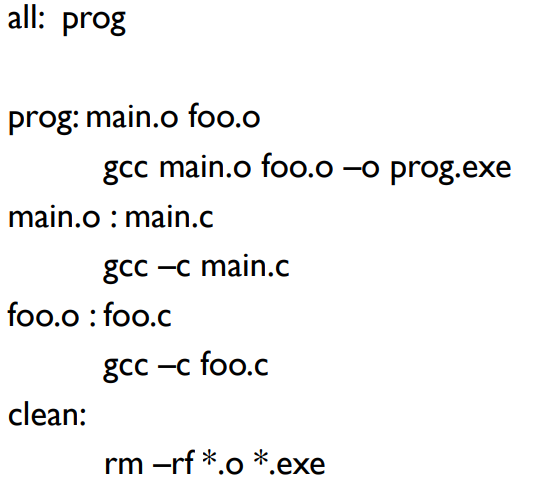
Автоматизация процесса сборки позволяет решить следующие задачи:

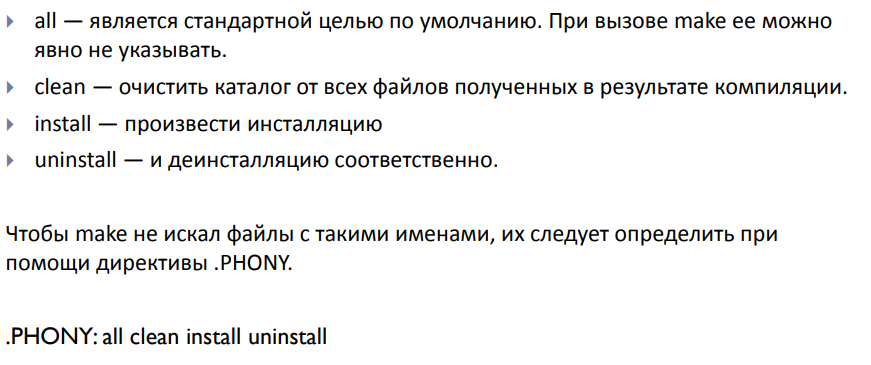
* освободить программистов от рутинной, нетворческой работы;
* ускорить процесс формирования исполнимого образа программного продукта и тем самым ускорить процесс разработки;
* уменьшить число ошибок, вызванных рассинхронизациями (типичным примером рассинхронизации можно считать ситуацию, когда в исполнимый образ по невнимательности программиста не попадает один из обновленных бинарных модулей, а специалист по тестированию обнаруживает связанные с этим ошибки);
* обеспечить наличие работоспособной версии кода проекта в произвольный момент времени (что особенно важно для проектов с открытым исходным кодом);
* минимизировать «плохие (некорректные)» сборки;
* устранить зависимость процесса сборки программного продукта от конкретного человека;
* обеспечить ведение истории сборок и релизов для разбора выпусков;

В совокупности все перечисленные выше возможности позволяют экономить время и деньги.

Мы используем makefile для автоматической сборки приложения.

Пример:





**30. Непрерывная интеграция: цели, основные принципы, инструментарий**

**Непрерывная интеграция** – это часть процесса разработки, в которой разрабатываемый проект собирается/тестируется в различных средах выполнения автоматически и непрерывно. Задумывалась данная методика для наиболее быстрого выявления ошибок/противоречий интеграции проекта, а соответственно снижения расходов на последующие простои.  
  
Принцип достаточно прост: на отдельной машине работает некая служба, в обязанности которой входит получение исходного кода проекта, его сборка, тестирование, логирование, а также возможность предоставить для анализа данные выполнения перечисленных операций.

Travis-ci — «хостинг непрерывной интеграции для open source сообщества».

Travis-ci работает с сервисом [github.com](http://github.com/), поэтому первым делом необходимо выложить туда исходный код своего проекта. За исключением мелких технических настроек (вроде установки хуков в репозитории), весь процесс сводится к описанию различных опций в файле .travis.yml. В нем нужно описать на каком языке работает проект, какие версии языка необходимо использовать среды окружения Данный файл необходимо разместить в корне github-репозитория.  
  
Если настройка прошла успешно, то travis-ci начинает непрерывно тестировать проект, отображая при этом текущий статус: красный цвет (возникли проблемы при тестировании), желтый (есть предупреждения) и зеленый (все тесты пройдены успешно). Помимо статуса можно увидеть: сообщение об ошибке или предупреждение, если что-то пошло не так; последний коммит и его автора; историю сборок и т.д. В целом интерфейс достаточно информативен и понятен. Помимо этого, travis-ci будет оповещать о проблемах по электронной почте.