1. Каскадная стратегия разработки: достоинства и недостатки
2. Инкрементальная стратегия разработки: достоинства и недостатки
3. Эволюционная стратегия разработки: достоинства и недостатки
4. Базовые стратегии разработки ПО, выбор модели
5. Гибкие технологии разработки ПО: Agile манифест
6. Экстремальное программирование (XP): тестирование и рефакторинг
7. Экстремальное программирование (XP): code style и метафора системы
8. Экстремальное программирование (XP): парное программирование и continuous integration
9. Экстремальное программирование (XP): соглашение о кодировании и коллективное владение кодом
10. Scrum: основные характеристики
11. Scrum: структура + роли
12. Scrum: структура + ритуалы
13. Scrum: структура + артефакты
14. Scrum: масштабируемость
15. Тестирование ПО: разработка теста для функции по ее спецификации
16. Тестирование ПО: статическое, динамическое тестирование
17. Тестирование ПО: методы белого и черного ящика
18. Системы контроля версий: история развития (RCS, CVS, SVN и git), локальные, централизованные и распределенные системы
19. Git: создание репозитория + фиксация изменений, стадии процесса
20. Git: создание ветки (branch)
21. Git: объединение веток (merge)
22. Git: конфликты и способы их разрешения
23. Git: работа с удаленным репозиторием (git fetch & git pull)
24. Git: работа с удаленным репозиторием (git rebase)
25. Git: работа с удаленным репозиторием (git push)
26. Git: модели ветвления
27. Системы автоматизации сборки: цели, задачи, примеры
28. Непрерывная интеграция: цели, основные принципы, инструментарий

№1

Каскадная стратегия разработки: достоинства и недостатки

Каскадная (водопадная) стратегия представляет собой однократный проход этапов разработки.

Данная стратегия основана на полном определении всех требований к разрабатываемому программному средству в начале процесса разработки. Возврат к уже выполненным этапам разработки не предусматривается. Промежуточные результаты в качестве версии программного средства не

распространяются.

/*Основными представителями моделей, реализующих каскадную стратегию, являются каскадная и V-образная модели.*/

Достоинства:

· Стабильность требований в течении ЖЦ (жизненный цикл)

· Необходимость только одного прохода этапов разработки, это обеспечивает простоту применения стратегии

· Простота планирования контроля и управления проектом

· Доступность для понимания заказчиков

Недостатки:

· Сложность полного формулирования требований

· Сложность структуры процесса разработки

· Непригодность промежуточных продуктов для использования

· Недостаточное участие пользователя в процессе разработки ПО

Области применения каскадной стратегии ограничены недостатками данной стратегии. Ее использование наиболее эффективно в следующих случаях:

1)при разработке проектов с четкими, неизменяемыми в течение ЖЦ требованиями, понятными реализацией и техническими методиками;

2)при разработке проекта, ориентированного на построение системы или продукта такого же типа, как уже разрабатывались разработчиками ранее;

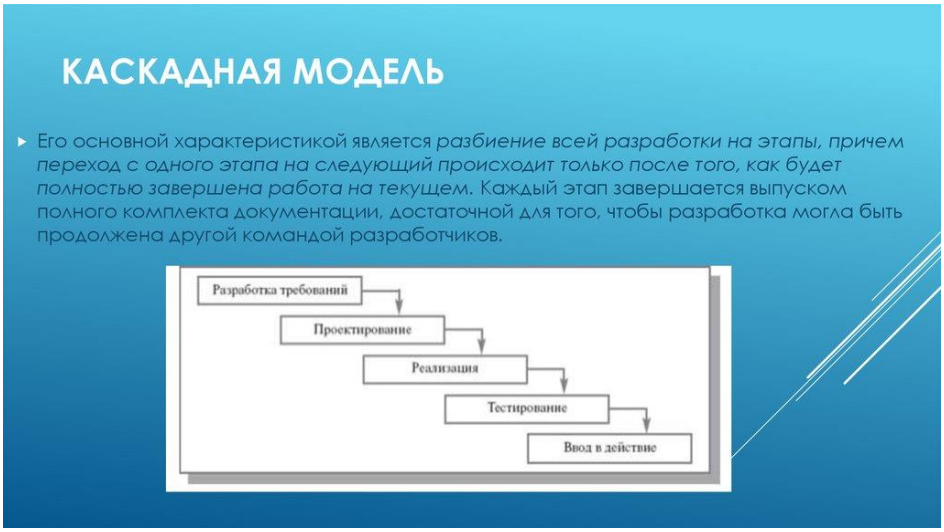
3)при разработке проекта, связанного с созданием и выпуском новой версии уже существующего продукта или системы;

4)при разработке проекта, связанного с переносом уже существующего продукта на новую платформу;

5) при выполнении больших проектов, которых задействовано несколько больших команд разработчиков

Данная модель применяется при разработке информационных систем, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования.

В каскадной модели легко управлять проектом. Благодаря её жесткости, разработка проходит быстро, стоимость и срок заранее определены. Но это палка о двух концах. Каскадная модель будет давать отличный результат только в проектах с четко и заранее определенными требованиями и способами их реализации. Нет возможности сделать шаг назад, тестирование начинается только после того, как разработка завершена или почти завершена. Продукты, разработанные по данной модели без обоснованного ее выбора, могут иметь недочеты (список требований нельзя скорректировать в любой момент), о которых становится известно лишь в конце из-за строгой последовательности действий.



№2

2.Инкрементальная стратегия разработки: достоинства и недостатки

Инкрементная стратегия представляет собой многократный проход этапов разработки с запланированным улучшением результата.

Данная стратегия основана наполном определении всех требований к разрабатываемому ПС в начале процесса разработки. Однако полный набор требований реализуется постепенно при последовательных циклах разработки.

Результат каждого цикла называется инкрементом. Различия между инкрементами соседних циклов в

ходе разработки постепенно уменьшаются.

Результат каждого цикла разработки может распространяться в качестве

очередной поставляемой версии ПС.

Особенностью инкрементной стратегии разработки является большое

количество циклов разработки при незначительной продолжительности цикла и небольших различиях между инкрементами соседних циклов.

/*Например, данная стратегия разработки программных средств и систем используется в компании Microsoft. Здесь на каждую версию ПС разрабатывается около тысячи инкрементов. Период разработки инкремента составляет один день. В ряде организаций используется недельный период разработки инкремента.*/

Инкрементная стратегия обычно основана на объединении элементов каскадной модели и прототипирования. При этом использование прототипирования позволяет существенно сократить продолжительность разработки каждого инкремента и всего проекта в целом.

Под прототипом понимается легко поддающаяся модификации и расширению рабочая модель предполагаемой системы(или программного средства), позволяющая пользователю получить представление о ключевых свойствах системы (или ПС) до ее полной реализации. Представителями моделей, реализующих инкрементную стратегию, являются, например, инкрементная и RAD-модель.

Преимущества инкрементной модели

· не требуется заранее тратить средства, необходимые для разработки всего проекта, поскольку сначала выполняется разработка и реализация основной функции или функции из группы высокого риска;

· в результате выполнения каждого инкремента получается функциональный продукт;

· правило по принципу "разделяй и властвуй" позволяет разбить возникшую проблему на управляемые части;

· возможность начать построение следующей версии проекта на переходном этапе предыдущей версии сглаживает изменения, вызванные сменой персонала;

· в конце каждой инкрементной поставки существует возможность пересмотреть риски, связанные с затратами и соблюдением установленного графика;

· требования стабилизируются (посредством включения в процесс пользователей) на момент создания определенного инкремента;

· улучшается понимание требований для более поздних инкрементов;

Недостатки инкрементной модели

· в модели не предусмотрены итерации в рамках каждого инкремента;

· определение полной функциональной системы должно осуществляться в начале жизненного цикла, чтобы обеспечить определение инкрементов;

· поскольку создание некоторых модулей будет завершено значительно раньше других, возникает необходимость в четко определенных интерфейсах;

· формальный критический анализ и проверку намного труднее выполнить для инкрементов, чем для системы в целом;

· для модели необходимы хорошее планирование и проектирование.

Область применения инкрементной модели

· если большинство требований можно сформулировать заранее;

· существует потребность быстро поставить на рынок продукт, имеющий функциональные базовые свойства;

· для проектов, на выполнение которых предусмотрен большой период времени разработки, как правило, один год;

· при равномерном распределении свойств различной степени важности;

· при разработке программ, связанных с низкой или средней степенью риска;

· при выполнении проекта с применением новой технологии;

· когда на ранних фазах оказывается, что самым оптимальным вариантом является применение принципа пофазовой разработки;

· когда однопроходная разработка системы связана с большой степенью риска.

№3

Эволюционная стратегия разработки: достоинства и недостатки

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. Как и при инкрементной стратегии, при реализации эволюционной стратегии зачастую используется прототипирование. В данном случае основной целью прототипирования является обеспечение полного понимания требований

Достоинства:

- возможность уточнения и внесения новых требований в процессе разработки;
- пригодность промежуточного продукта для использования;
- возможность управления рисками;
- обеспечение широкого участия пользователя в проекте, начиная с ранних этапов, что минимизирует возможность разногласий между заказчиками и разработчиками и обеспечивает создание продукта высокого качества;

 реализация преимуществ каскадной и инкрементной стратегий.

Недостатки:

 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;
 сложность планирования и управления проектом;
 необходимость активного участия пользователей в проекте, что реально не всегда осуществимо;
 необходимость в мощных инструментальных средствах и методах прототипирования;
 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.

№4

Базовые стратегии разработки ПО, выбор модели

Базовые стратегии разработки ПО

Начальный этап:

кодирование – устранение ошибок

Три базовые стратегии:

- каскадная;
- инкрементная;
- эволюционная.

Определяются характеристиками:

- проекта;
- требований к продукту;
- команды разработчиков;
- команды пользователей.

Каскадная модель (*waterfall model)*

Каскадная модель

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.

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.
- Первый инкремент реализует базовые функции программного средства. В последующих инкрементах функции программного средства постепенно расширяются, пока не будет реализован весь набор требований. Различия между инкрементами соседних циклов в ходе разработки постепенно уменьшаются.
- Результат каждого цикла разработки может распространяться в качестве очередной поставляемой версии программного средства или системы.

№5

Система контроля версий (СКВ)

Распределение СКВ позволяет полностью копировать репозиторий с возможностью быть скопированным на другой сервер для продолжения работы. Каждая копия является полным бэкапом. Они способны одновременно воздействовать с удалёнными репозиториями, то есть можно применять несколько подходов в разработке.

Локальные системы контроля версий

RCS(revision control system) была разработана в начале 1980-х годов Вальтером Тичи (Walter F. Tichy).

- позволяет хранить только файл; управление несколькими происходит вручную;
- неудобный механизм одновременной работы нескольких пользователей с системной;

$ci file.txt /*check in, т.е регистрировать*/

$ co file.txt /*check out, проверить*/

Централизированные СКВ

CVS (Concurrent Version 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.

- позволяет хранить много файлов в проекте;
- нет возможности сохранять версии дирректорий;
- перемещение, или переименование файлов не подвержено системе контроля версий(если при заливке кто-то исправит файл, то тот же у другого «слетит»);
- использует клиент-сервер – архитектуру, в которой вся информация о версиях хранится на сервере;

$cvs checkout path-in-repository /*co=checkout*/
$cvs commit –m ‘Some changes’
$cvs update

SVN(Subversion, Свободная централизованная система контроля версий) свободная централизованная система управления версиями, официально выпущенная в 2004 году компанией CollabNet

- ✓ автомарные внесения изменений (commit). В случае, если обработка коммита была прервана, не будет внесено никаких изменений;
- ✓ переименование, копирование и перемещение файлов – сохраняет всю историю изменений;
- ✓ дирректории, символические ссылки и мета-данные – всё подвержено системе контроля версий;
- ✓ эффективное хранение изменений для бинарных файлов;

$svn checkout-in-repository
$svn commit-in “Some changes”
$svn up

Распределённые СКВ

Git (2005, **Линус Торвальдс**) /*на сладенькое https://habr.com/ru/post/374887/*/

- ❖ клиенты могут выгружать последние версии файлов, копировать весь репозиторий;
- ❖ есть возможность восстановления «мёртвого сервера»;
- ❖ скорость;
- ❖ простота дизайна;
- ❖ поддержка нелинейной разработки (тысячи параллельных веток);
- ❖ полная распределённость:
- ❖ возможность эффективной работы с Linux:

$git <verb> -help
$git init
$git add <filename>

№6

Экстремальное программирование (XP): тестирование и рефакторинг.

Экстрема́льное программи́рование — одна из гибких методологий разработки программного обеспечения. Авторы методологии — Кент Бек, Уорд Каннингем, Мартин Фаулер и другие.

Название методологии исходит из идеи применить полезные традиционные методы и практики разработки программного обеспечения, подняв их на новый «экстремальный» уровень.

Тестирование

XP предполагает написание автоматических тестов. Особое внимание уделяется двум разновидностям тестирования:

- юнит-тестирование модулей;
- функциональное тестирование.

Разработчик не может быть уверен в правильности написанного им кода до тех пор, пока не сработают абсолютно все тесты модулей разрабатываемой им системы. Тесты модулей позволяют разработчикам убедиться в том, что каждый из них по отдельности работает корректно. Они также помогают другим разработчикам понять, зачем нужен тот или иной фрагмент кода, и как он функционирует — в ходе изучения кода тестов логика работы тестируемого кода становится понятной, так как видно, как он должен использоваться. Тесты модулей также позволяют разработчику без каких-либо опасений выполнять рефакторинг.

Функциональные тесты предназначены для тестирования функционирования логики, образуемой взаимодействием нескольких частей. Они менее детальны, чем юнит-тесты, но покрывают гораздо больше. По этой причине в промышленном программировании написание функциональных тестов нередко имеет больший приоритет, чем написание юнит-тестов.

Для XP более приоритетным является подход, называемый TDD (*test-driven development* — разработка через тестирование). В соответствии с этим подходом сначала пишется тест, который изначально не проходит (так как логики, которую он должен проверять, ещё просто не существует), затем реализуется логика, необходимая для того, чтобы тест прошёл. TDD, в некотором смысле, позволяет писать код, более удобный в использовании — потому что при написании теста, когда логики ещё нет, проще всего позаботиться об удобстве будущей системы.

Рефакторинг

Рефакторинг — это методика улучшения кода без изменения его функциональности. XP подразумевает, что однажды написанный код в процессе работы над проектом почти наверняка будет неоднократно переделан. Разработчики XP безжалостно переделывают написанный ранее код для того, чтобы улучшить его. Этот процесс называется рефакторингом. Отсутствие тестового покрытия провоцирует отказ от рефакторинга в связи с боязнью поломать систему, что приводит к постепенной деградации кода.

№7

Экстремальное программирование (XP): code style и метафора системы

Экстремальное программирование(XP)- название методологии исходит из идеи применить полезные традиционные методы и практики разработки программного обеспечения, подняв их на новый «экстремальный» уровень

Стандарт оформления кода (code style) — это набор определенных правил при написании исходного кода, на определенном языке программирования.

Метафора системы (system metaphor). Хорошая метафора системы означает простоту именования классов и переменных. В реальной жизни поиск метафоры — крайне сложное занятие; найти хорошую метафору непросто. В любом случае команда должна иметь единые правила именования.

№8

Экстремальное программирование (XP): парное программирование и continuous integration

Парное программирование

Весь код создается парами программистов, работающих за одним компьютером. Один из них работает непосредственно с текстом программы, другой просматривает его работу и следит за общей картиной происходящего.

Непрерывная интеграция

Если выполнять интеграцию разрабатываемой системы достаточно часто, то можно избежать большей части связанных с ней проблем. В XP интеграция кода всей системы выполняется несколько раз в день, после того, как разработчики убедились в том, что все тесты модулей корректно срабатывают.

№9

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

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

Коллективное владение кодом

Коллективное владение кодом стимулирует разработчиков подавать идеи для всех частей проекта, а не только для своих модулей. Любой разработчик может изменять любой код для расширения функциональности, исправления ошибок или рефакторинга.

С первого взгляда это выглядит как хаос. Однако принимая во внимание что как минимум любой код создан парой разработчиков, что Unit тесты позволяют проверить корректность внесенных изменений и что в реальной жизни все равно так или иначе приходится разбираться в чужом коде, становится ясно что коллективное владение кодом значительно упрощает внесение изменений и снижает риск связанный с высокой специализацией того или иного члена команды.

№10

Scrum: 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”

№11

11. Scrum: структура + роли

Структура

1. **Роли**
    a. Владелец продукта

b. Скрам-Мастер
c. Команда

2. **Ритуалы**
   a. Планирование спринта
   b. Обзор спринта
   c. Спринт ретроспектива
   d. Ежедневный Скрам
3. **Артефакты**
   a. Бэклог продукта
   b. Спринт бэклог
   c. Burndown charts

Роли

1) **Владелец продукта**
   a) Один человек
   b) Определяет требования продукта
   c) Определяет дату релиза и наполненность
   d) Ответственный за доходность проекта (ROI)
   e) Приоритезирует требования, исходя из их рыночной ценности
   f) Корректирует приоритеты на каждой итерации, если необходимо
   g) Принимает работу
2) **Скрам-мастер**
   a) Представляет руководство проекта
   b) Ответственен за внедрение ценностей и практик Скрам
   c) Не раздает задания
   d) Устраняет препятствия
   e) Ответственен за эффективность работы команды
   f) Обеспечивает видимость и прозрачность ситуации в команде
   g) Защищает команду от внешних воздействий
3) **Команда**
   a) Обычно 5-9 человек
   b) Кросс функциональная (программисты, тестировщики, дизайнеры…)
   c) Заняты полный день (могут быть исключения, например администратор базы данных)
   d) Команды самоорганизуются (в идеале, нет специальных ролей)
   e) Состав команды может меняться только между спринтами

№12

«Scrum: структура + ритуалы»

Практика работы в команде по методологии скрам состоит из целого ряда ритуалов. Ритуалы отмечают ключевые события в процессе выполнения спринта. Обязанность скрам-мастера провести каждый ритуал и убедится, что они сфокусированы на своих целях и добиваются желаемых результатов на благо всей команды.

Что такое ритуалы?

Каждый ритуал — это встреча лицом к лицу в реальном времени, что позволяет отвлечься от непосредственной работы и дает возможность целевого общения с коллегами о смысле этой работы. Скрам ставит коммуникацию выше документации, именно поэтому он обеспечивает регулярность встреч с четко определенными возможностями различных типов полезного общения лицом к лицу.

Ритуалы:

Планирование спринта (Sprint Planning Meeting)

Происходит в начале новой итерации спринта.

Из бэклога проекта выбираются задачи, обязательства по выполнению которых за спринт принимает на себя команда;

На основе выбранных задач создается бэклог спринта. Каждая задача оценивается в идеальных человеко-часах;

Решение задачи не должно занимать более 12 часов или одного дня. При необходимости задача разбивается на подзадачи;

Обсуждается и определяется, каким образом будет реализован этот объем работ;

Продолжительность митинга ограничена сверху 8 часами в зависимости от продолжительности спринта, опыта команды и т. п.

(первая часть совещания), участвуют скрам-мастер, владелец продукта и скрам-команда: выбирают задачи из бэклога продукта;

(вторая часть совещания), участвует только скрам-команда: обсуждают технические детали реализации, наполняют бэклог спринта.

Ежедневное стоячее SCRUM-совещание (Daily SCRUM)

- начинается в одно и то же время в одном месте;
- все могут наблюдать, но только «свиньи» говорят;
- в митинге участвуют SCRUM Master, SCRUM Product Owner и SCRUM Team;
- длится ровно 15 минут;
- все участники во время Daily SCRUM стоят (митинг в формате Daily Standup).

SCRUM-мастер задает каждому члену SCRUM-команды три вопроса:

- что я сделал с момента прошлой встречи для того, чтобы помочь команде разработки достигнуть цели спринта?
- что я сделаю сегодня для того, чтобы помочь команде разработки достичь цели спринта?
- вижу ли я препятствия для себя или команды разработки, которые могли бы затруднить достижение цели спринта? (Над решением этих проблем методом фасилитации работает скрам-мастер. Обычно это решение проходит за рамками ежедневного совещания и в составе лиц, непосредственно затронутых данным препятствием.)

Обзор итогов спринта (Sprint review meeting)

Проводится в конце спринта.

- Команда демонстрирует прирост инкремента продукта всем заинтересованным лицам.
- Все члены команды участвуют в демонстрации (один человек на демонстрацию или каждый показывает, что сделал за спринт).
- Нельзя демонстрировать незавершенную функциональность.
- Ограничена четырьмя часами в зависимости от продолжительности итерации и прироста функциональности продукта.

Ретроспективное совещание (Retrospective meeting)

Проводится в конце спринта.

- Члены скрам-команды, скрам-мастер и продукт-оунер высказывают свое мнение о прошедшем спринте.
- Скрам-мастер задает два вопроса всем членам команды:
    - Что было сделано хорошо в прошедшем спринте?
    - Что надо улучшить в следующем?
- Выполняют улучшение процесса разработки (обсуждают варианты решения проблем, фиксируют удачные решения и вызвавшегося владельца продукта).
- Ограничена четырьмя часами для спринта любой длины.

№13

13.Scrum - Артефакты

Scrum Artifacts предоставляют ключевую информацию, которую команда Scrum и заинтересованные стороны должны знать о понимании разрабатываемого продукта, о выполненных мероприятиях и планируемых мероприятиях в проекте. В Scrum Process Framework определены следующие артефакты:

- Product Backlog
- Sprint Backlog
- Burn-Down Chart
- Increment

Это минимально необходимые артефакты в проекте схватки, а артефакты проектов не ограничены ими.

Продукт Backlog - это упорядоченный список функций, которые необходимы как часть конечного продукта, и это единственный источник требований для любых изменений, которые должны быть внесены в продукт.

Sprint Backlog - это прогноз команды о том, какая функциональность будет доступна в следующем Приращении и в работе, необходимой для реализации этой функциональности в качестве рабочего продукта Increment.

Инкремент - это сумма всех элементов отставания продукта, выполненных во время спринта, в сочетании с приращениями всех предыдущих Спринтов. В конце Sprint новый Increment должен быть рабочим продуктом, что означает, что он должен быть в работоспособном состоянии. Он должен находиться в рабочем состоянии, независимо от того, решает ли его владелец выпускать его.
Sprint Burn-Down Chart - это практика, направленная на то, чтобы расходовать работу, затрачиваемую командой Scrum. Это было доказано, что это полезный метод для мониторинга прогресса Sprint в направлении Sprint.

№14

Scrum: масштабируемость

/*SCRUM (англ. *scrum*«схватка») — метод управления проектами. SCRUM (SCRibing Unified Methodology) — набор принципов, ценностей, политик, ритуалов, артефактов, основанных на скрайбинге и скрапбукинге, на которых строится процесс SCRUM-разработки, позволяющий в жестко фиксированные и небольшие по времени итерации, называемые спринтами (sprints), предоставлять конечному пользователю работающий продукт с новыми бизнес-возможностями, для которых определен наибольший приоритет.

SCRUM используется как в сфере разработки ПО, так и в других производственных бизнес-отраслях.

Кроме управления проектами по разработке ПО, SCRUM может также использоваться в работе команд поддержки программного обеспечения, как подход к управлению разработкой и сопровождению программ, а также в ремонте.*/

Масштабирование Scrum.

Что важно знать, прежде чем задумываться о масштабировании

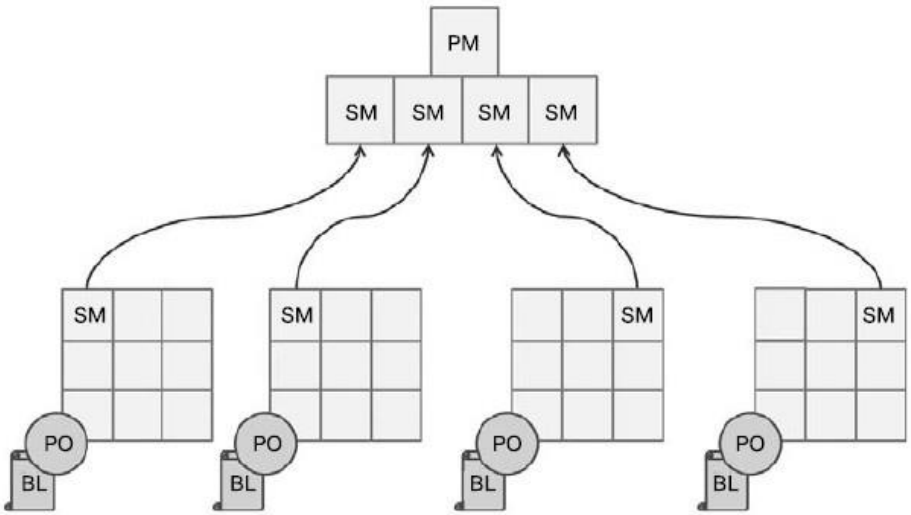
Когда у Scrum Master-а возникают мысли о масштабировании, это говорит о том, что команда столкнулась с тремя ключевыми проблемами, которые нужно как можно скорее решить.

1. Стало сложно планировать спринт: Product Owner не справляется с составлением требований для всех, поскольку команда слишком большая. Также он слишком длинный.
2. Ретроспектива теперь больше похожа на зомби-апокалипсис: одна половина команды пытается перекричать другую, а Scrum Master изо всех сил старается контролировать ситуацию, но получается плохо.
3. Постоянно что-то «впихивается» в текущий спринт, хотя, на самом деле, с дополнительным объемом задач справиться невозможно.

6 советов по масштабированию небольших команд.

1. Разбить Scrum Team на маленькие команды с полным набором скиллов (минимальная команда может состоять из двух человек, хоть Scrum Guide и советует начинать с трех).
2. Настроить полностью автоматизированный workflow в Jira (или любом другом трекере).
3. Команда должна слаженно работать даже в отсутствие Scrum Master-а. Главная цель любого Scrum Master-а — сделать так, чтобы команда отлично работала без его активного участия в процессе, особенно когда дело касается масштабируемых проектов.
4. Обратить внимание на Self Selection process — процесс, когда участники команд сами выбирают, в какой части продукта они хотят работать. Эта практика хороша для поднятия боевого духа и мотивации сотрудников.
5. Разделить команду на самодостаточные блоки, по которым они закрывают те или иные проблемы. Можете разбить большой продукт на функциональные куски и сформировать команды, которые будут работать в разном окружении. Здесь как раз и пригодится Self Selection process.
6. Не давайте сотрудникам потерять ощущение, что без их работы мир рухнет! Нет более или менее важных команд. Все крутые и все равны!

Небольшие команды чаще показывают хорошие результаты, чем большие, поэтому необходимо по возможности вести разработку компактными командами. К сожалению, часто бывает, что объем проекта и сроки его реализации просто не позволяют вести разработку 5–9 людьми и приходится задействовать несколько команд.



Привлечение нескольких команд:

Для организации разработки больших проектов или портфеля проектов необходимо масштабировать Scrum на следующий уровень. Со стороны команд разработки это выливается в проведение Scrum of Scrum.

На этот митинг собираются скрам-мастера (SM) в качестве представителей конкретных команд. Организует собрание руководитель программы (Program Manager, PM). При использовании дивизионной организационной структуры на данном уровне он также может являться руководителем соответствующего дивизиона (подразделения).

В качестве базовой структуры митинга можно предложить каждому скрам-мастеру ответить на следующие вопросы.

1. Что было сделано с прошлого Scrum of Scrum?
2. Какие были проблемы?
3. Что будет сделано к следующему Scrum of Scrum?

При этом акцент нужно сделать на проблемах, которые команда не может решить сама и вынуждена передавать выше

№15

Тестирование ПО: разработка теста для функции по её спецификации

Тести́рование програ́ммного обеспе́че́ния — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом.(или процесс, имеющий целью выявление ситуаций, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации).

Тест — это выполнение определенных условий и действий, необходимых для проверки работы тестируемой функции или её части.

Цели:

- Найти ошибки
- Добиться отсутствия ошибок (отладка)

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.

Такой подход позволяет декларативно создавать тесты. Спецификации легко читаются и дополняются при изменении требований. Тестовый код получается компактным. Его легко поддерживать и расширять. Этот Файл состоит из:

- Комментариев в начале файла. Комментарии могут состоять из любых символов кроме $.
- Проверяемых свойств. Имя каждого свойства начинается с символа $. Конец значения свойства определяется либо началом следующего свойства, либо концом файла.

Уровни тестирования:

- Тестирование компонентов — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто тестирование компонентов осуществляется разработчиками программного обеспечения.
- Интеграционное тестирование — тестируются интерфейсы между компонентами, подсистемами или системами. При наличии резерва времени на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.
- Системное тестирование — тестируется интегрированная система на её соответствие требованиям.
    - Альфа-тестирование — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приёмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программа.
    - Бета-тестирование — в некоторых случаях выполняется распространение предварительной версии (в случае проприетарного программного обеспечения иногда с ограничениями по функциональности или времени работы) для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Часто для свободного и открытого программного обеспечения стадия альфа-тестирования характеризует функциональное наполнение кода, а бета-тестирования — стадию исправления ошибок. При этом как правило на каждом этапе разработки промежуточные результаты работы доступны конечным пользователям.

1. Программа и требования поступают к тестировщику.
2. Он совершает необходимые операции, ведет наблюдение за тем, как ПО выполняет поставленные задачи.
3. По результатам проверки формируется список соответствий и несоответствий.
4. Используя полученную информацию, можно либо улучшить существующий софт, либо обновить требования к разрабатываемой программе.

/*

1. **Юзабилити-тестирование** *(проверка эргономичности)*помогает определить: удобен ли сайт или пользовательский интерфейс для его предполагаемого применения.
2. **Создание чек-листа** — подготовка набора тестов, внесение необходимых предложений в разрабатываемые требования (с точки зрения качества).
3. **Тестирование.** Получив готовую для проверки программу или её часть, специалист проверяет её соответствие требованиям на выбранном наборе тестов. В случае обнаружения дефектов — передаёт разработчикам набор задач, необходимых для улучшения продукта до состояния соответствия требованиям.
4. **Верификация** — проверка, которая показывает: были ли исправлены ошибки, обнаруженные в результате тестов.
5. **Тест производительности** *(performance testing)* проводится на стендах, где в дальнейшем будет эксплуатироваться софт. Цель — выявление проблем стенда (не софта), имитация работы пользователей, проверка на стрессоустойчивость. Позволяет убедиться, что приложение/система справится с реальной нагрузкой в будущем.

№16

Тестирование ПО: статическое, динамическое тестирование

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. (Верификация проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих артефактов и процессов их разработки правилам и стандартам.) 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. (Валидация проверяет соответствие любых создаваемых или используемых в ходе разработки и сопровождения ПО артефактов нуждам и потребностям пользователей и заказчиков этого ПО, с учетом законов предметной области и ограничений контекста использования ПО.)

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.).

№17

Тестирование ПО: методы белого и черного ящика

Тестирование по стратегии белого ящика

Тестирование по стратегии белого ящика — тестирование кода на предмет логики работы программы и корректности её работы с точки зрения компилятора того языка, на котором она писалась.Тестирование по стратегии белого ящика, также называемое техникой тестирования, управляемой логикой программы, позволяет проверить
внутреннюю структуру программы. Исходя из этой стратегии тестировщик получает тестовые данные путем анализа логики работы программы.
Техника Белого ящика включает в себя следующие методы тестирования:

1. покрытие решений
2. покрытие условий
3. покрытие решений и условий
4. комбинаторное покрытие условий

Тестирование чёрного ящика или поведенческое тестирование

Тестирование чёрного ящика или поведенческое тестирование — стратегия (метод) тестирования функционального поведения объекта (программы, системы) с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве тестируемого объекта. Под стратегией понимаются систематические методы отбора и создания тестов для тестового набора. Стратегия поведенческого теста исходит из технических требований и их спецификаций.

№18

Гибкие технологии разработки ПО: Agile манифест

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.
Методики: экстремальное программирование, DSDM, Scrum, FDD.

Большинство гибких методологий нацелены на минимизацию рисков путём сведения разработки к серии коротких циклов – итерациями, которые обычно длятся две-три недели. Каждая итерация сама по себе выглядит как программный проект в миниатюре и включает все задачи, необходимые для выдачи мини-прироста по функциональности*: планирование, анализ требований, проектирование, программирование, тестирование и документирование.*

Манифест гибкой разработки программного обеспечения (англ. Agile Manifesto) — основной документ, содержащий описание ценностей и принципов гибкой разработки программного обеспечения, разработанный в феврале 2001 года на встрече 17 независимых практиков нескольких методик программирования, именующих себя «Agile Alliance».

- **Люди и взаимодействие** важнее процессов и инструментов
- **Работающий продукт** важнее исчерпывающей документации
- **Сотрудничество с заказчиком** важнее согласования условий контракта
- **Готовность к изменениям** важнее следования первоначальному плану

Принципы Agile-манифеста:

- ✓ Наивысшим приоритетом для нас является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения
- ✓ Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.
- ✓ Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.
- ✓ На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
- ✓ Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
- ✓ Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.
- ✓ Работающий продукт — основной показатель прогресса.
- ✓ Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.
- ✓ Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
- ✓ Простота — искусство минимизации лишней работы — крайне необходима.
- ✓ Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
- ✓ Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

№19

Создание Git-репозитория

Для создания Git-репозитория существуют два основных подхода. Первый подход — импорт в Git уже существующего проекта или каталога. Второй — клонирование уже существующего репозитория с сервера.

Создание репозитория в существующем каталоге

Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в проектный каталог и в командной строке ввести

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. (В главе 9 приведено подробное описание файлов содержащихся в только что созданном вами каталоге `.git`)

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это вы можете с помощью нескольких команд `git add` указывающих индексируемые файлы, а затем `commit`:

```
$ git add *.c

$ git add README

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

Мы разберём, что делают эти команды чуть позже. На данном этапе, у вас есть Git-репозиторий с добавленными файлами и начальным коммитом.

Клонирование существующего репозитория

Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда `git clone`. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется `clone`, а не `checkout`. Это важное отличие — Git получает копию практически всех данных, что есть на сервере. Каждая версия каждого файла из истории проекта забирается (pulled) с сервера, когда вы выполняете `git clone`. 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 (вы можете потерять часть серверных перехватчиков (server-side hooks) и т.п., но все данные, помещённые под версионный контроль, будут сохранены, подробнее см. в главе 4).
Клонирование репозитория осуществляется командой `git clone [url]`. Например, если вы хотите клонировать библиотеку Ruby Git, известную как Grit, вы можете сделать это следующим образом:

```
$ git clone git://github.com/schacon/grit.git
```

Эта команда создаёт каталог с именем `grit`, инициализирует в нём каталог `.git`, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новый каталог `grit`, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от `grit`, можно это указать в следующем параметре командной строки:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mygrit.

№20

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

№21

Git: объединение веток (merge)

Слияние — обычная практика для разработчиков, использующих системы контроля версий. Независимо от того, созданы ли ветки для тестирования, исправления ошибок или по другим причинам, слияние фиксирует изменения в другом месте. Слияние принимает содержимое ветки источника и объединяет их с целевой веткой. В этом процессе изменяется только целевая ветка. История исходных веток остается неизменной.

Плюсы:

1)простота;

2)сохраняет полную историю и хронологический порядок; поддерживает контекст ветки.

Минусы:

1)история коммитов может быть заполнена (загрязнена) множеством коммитов;

2)Git merge создает новый «Merge commit» , который содержит историю обеих веток

№22

Git: конфликты и способы их разрешения

Иногда процесс не проходит гладко. Если вы изменили одну и ту же часть одного и того же файла по-разному в двух объединяемых ветках, Git не сможет их чисто объединить.

 $ git merge iss53 Auto-merging index.html CONFLICT (content): Merge conflict in index.html Automatic merge failed; fix conflicts and then commit the result.

Git не создал коммит слияния автоматически. Он остановил процесс до тех пор, пока вы не разрешите конфликт.

 $ git status On branch master You have unmerged paths. (fix conflicts and run "git commit")

Unmerged paths: (use "git add <file>..." to mark resolution)

  both modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

Все, где есть неразрешенные конфликты слияния, перечисляется как неслитое. Git добавляет в конфликтующие файлы стандартные пометки разрешения конфликтов, чтобы вы могли вручную открыть их и разрешить конфликты.

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Разрешив каждый конфликт во всех файлах, запустите git add для каждого файла, чтобы отметить конфликт как решенный. Подготовка (staging) файла помечает его для Git как разрешенный конфликт.

№23

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

Для получения данных из удалённых проектов, следует выполнить:

$ git fetch [remote-name]

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 git 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.
Если у вас есть ветка, настроенная на отслеживание удалённой, то вы можете использовать команду:
$ git pull [remote-name] = git fetch + git merge
чтобы автоматически получить изменения из удалённой ветви и слить их со своей текущей ветвью. 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.

№24

«Git: работа с удаленным репозиторием (git rebase)»

Git'e есть два способа включить изменения из одной ветки в другую: merge (слияние) и rebase (перемещение).

Итак git работает с комитами. Каждый комит — набор изменений. У каждого комита есть уникальный hash. Когда происходит слияние веток посредством merge:

git merge "another_branch"

то все комиты сохраняются — сохраняются комментарии комита, его hash + как правило добавляется еще один искусственный комит. При этом комиты могут чередоваться друг с другом. Это не всегда удобно. Допустим ваш комит решили откатить — выискивать в общем списке где ваш комит, а где не ваш не очень приятно. И вообще — в общей истории хочется видеть действительно важные изменения, а не «ой, я забыл поставить ;».

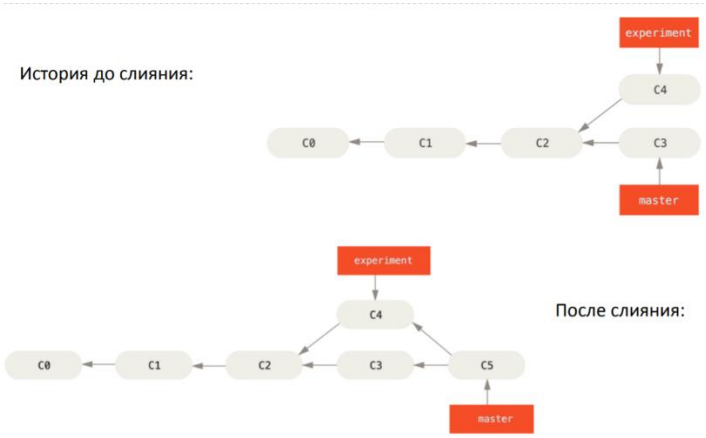
Для того, чтобы несколько комитов склеивать в один можно использовать rebase.

Хотя в интерфейсе GitHub есть кнопочка squash & commit — это когда вы создаете pull request (PR) из одной ветки в другую (как правило из вашей рабочей ветки в основную) и после прохождения всех формальностей можно нажать squash & commit, обновить комментарий и ваши изменения появятся в основной ветке как один комит.

Осторожно rebase может менять hash комита и приводить к конфликтам слияний, особенно если над одной веткой трудятся несколько человек.

Основы перемещения

Если мы вернёмся назад к одному из ранних примеров из раздела про слияние (см. рис. 3-27), увидим, что мы разделили свою работу на два направления и сделали коммиты на двух разных ветках.



Перемещение работает следующим образом: находится общий предок для двух веток (на которой вы находитесь сейчас и на которую вы выполняете перемещение); для каждого из коммитов в текущей ветке берётся его дельта и сохраняется во временный файл; текущая ветка устанавливается на тот же коммит, что и ветка, на которую выполняется перемещение; и, наконец, одно за другим применяются все изменения.

№25

Git: работа с удаленным репозиторием (git push)

Отображение удалённых репозиториев

Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду `git remote`

Добавление удалённых репозиториев

В предыдущих разделах мы упомянули и немного продемонстрировали добавление удалённых репозиториев, сейчас мы рассмотрим это более детально. Чтобы добавить новый удалённый Git-репозиторий под именем-сокращением, к которому будет проще обращаться, выполните `git remote add [сокращение] [url]`:
Теперь вы можете использовать в командной строке имя pb вместо полного URL. Например, если вы хотите извлечь (fetch) всю информацию, которая есть в репозитории Павла, но нет в вашем, вы можете выполнить `git fetch pb`

Push

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая: `git push [удал. сервер] [ветка]`. Чтобы отправить вашу ветку master на сервер `origin` (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

Инспекция удалённого репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду `git remote show [удал. сервер]`. Если вы выполните эту команду с некоторым именем, например, `origin`, вы получите что-то подобное:

Удаление и переименование удалённых репозиториев

Для переименования ссылок в новых версиях Git'a можно выполнить `git remote rename`, это изменит сокращённое имя, используемое для удалённого репозитория. Например, если вы хотите переименовать `pb` в `paul`, вы можете сделать это следующим образом:

№26

Git: модели ветвления

https://gist.github.com/jbenet/ee6c9ac48068889b0912

https://nvie.com/posts/a-successful-git-branching-model/

№27

Системы автоматизации сборки: цели, задачи, примеры

Makefile

цель : зависимости

[tab] команды

$ gcc main.c foo.c –o prog.exe

// Makefile

all: gcc main.c foo.c –o prog.exe

Цели

all — является стандартной целью по умолчанию. При вызове make ее можно явно не указывать.

clean — очистить каталог от всех файлов полученных в результате компиляции.

install — произвести инсталляцию uninstall — и деинсталляцию соответственно.

Чтобы make не искал файлы с такими именами, их следует определить при помощи директивы .PHONY.

Задачи

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;

Примеры

Makefile#1

```
all:  prog

prog: main.o foo.o
        gcc main.o foo.o –o prog.exe
main.o : main.c
        gcc –c main.c
foo.o : foo.c
        gcc –c foo.c clean:  rm –rf *.o *.exe
```

Makefile#2

```
CC = gcc
CFLAGS = -c –Wall

main.o : main.c
        $(CC) $(CFLAGS) main.c
foo.o : foo.c
        $(CC) $(CFLAGS) foo.c
```

№28

Непрерывная интеграция: цели, основные принципы, инструментарий

Непрерывная интеграция (continuous integration). Интеграция новых частей системы должна происходить как можно чаще, как минимум раз в несколько часов. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

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.

Continuous integration

1) На выделенном сервере организуется служба, в задачи которой входят:
   a) Получение исходного кода из репозитория;
   b) Сборка проекта;
   c) Выполнение тестов;
   d) Развертывания готового проекта;
   e) Отправка отчетов.
2) Локальная сборка может осуществляться:
   a) По внешнему запросу;
   b) По расписанию;
   c) По факту обновления репозитория и по другим критериям.