Описание

Курс будет состоять из двух частей. В первой части занятия будут проходить в виде семинаров с докладами студентов по предложенным темам. Для успешного прохождения первой половины курса необходимо сделать как минимум три доклада по предложенным темам. Во второй части курса, студентам будет предложено разделиться на комманды по 3-4 человека, для выполнения практического задания, а именно разработки компьютерной игры “Балда” с применением методологии Scrum.

Зачет получают те студенты, которые успешно выполнили задания первой половины курса (не менее 3-х докладов) и успешно реализовали все задания второй половины курса (наличие работающей компьютерной игры “Балда”)

Требования к отчетам

Для практических занятий, отчет должен содержать слайды презентации и текст доклада. Для лабораторных работ, содержание отчета будет указано в задании.

Практическое задание №1

“История программной инженерии”

Цель: Знакомство с историей возникновения и развития программной инженерии.

Задание: Подготовить доклад о людях которые на Ваш взгляд внесли весомый вклад в развитие и становление программной инженерии. Например: Керниган Ричи, Бьерн Страуструп и др.

Требования к докладу: Доклад должен сопровождаться презентацией. Длительность доклада не должна превышать 20 минут. Доклады подготавливаются индивидеально. У двух человек в группе не может быть одинакового доклада.

Практическое задание №2

“История развития языков программирования”

Цель: Знакомство с историей возникновения и развития языков программирования.

Задание: Подготовить доклад о языках программирования которые на Ваш взгляд имели или имеют огромное влияние на развитие программной инженерии. Например: Java, Python и др. Особое внимание стоит уделить сфере их применения, а также истории развития.

Практическое задание №3
“Профессиональные сообщества”

Цель: Знакомство с историей возникновения и развития профессиональных сообществ программистов. Зачем они нужны, какие задачи решают.

Задание: Подготовить доклад о професиональных сообществах программистов, которые на Ваш взгляд имели или имеют влияние на развитие программной инженерии.

Практическое задание №4
“Сертификация”

Цель: Познакомится с видами сертификации специалистов в программной инженерии.

Задание: Подготовить доклад о различных сертификационных программах для программистов. Рассказать о их назначении и условиях сертификации.

Лабораторная работа №1

Централизированная система контроля версий

“Subversion (SVN)”

Цель: Получить навыки нароты с системой контроля версий “Subversion”

При разработке проектов программисты сталкиваются с рядом трудностей, например:

1. Необходимость хранения различный версий програмного продукта(ПП). Каждую версию можно сохранять в отдельную папку, но со временем станет достаточно сложно управляться с большим количеством файлов.
2. Если над программой работает сразу несколько человек, необходимо объединять сделанные ими изменения.

Для решения этих проблем удобно использовать системы контроля версий, например Subversion (SVN).

Обычно система контроля версий состоит из двух частей:

- **Сервер**, или репозиторий — где хранятся все исходные коды программы, а также история их изменения.
- **Клиент**. Каждый клиент имеет свою локальную копию (working copy) исходных кодов, с которой работает разработчик.

В связи с тем, что разработчики работают только с локальными копиями, могут возникать трудности, когда два и более человек изменяют один и тот же файл, что приводит к конфликтам.

В системах контроля версий есть две модели разрешения конфликтов:

- **Блокировка — изменение — разблокировка.** Согласно этой модели, когда кто-либо начинает работу с файлом, этот файл блокируется, и все остальные пользователи теряют возможность его редактирования. Очевидным недостатком такой модели является то, что файлы могут оказаться надолго заблокированными, что приводит к простоям в разработке проекта.
- **Копирование — изменение — слияние.** В данной модели каждый разработчик свободно редактирует свою локальную копию файлов, после чего выполняется слияние изменений. Недостаток этой модели в том, что может возникать необходимость разрешения конфликтов между изменениями файла.

В SVN доступны обе модели, причём вторая является основной. В то же время для некоторых типов файлов (например, изображения) целесообразно использовать первую.

Subversion (SVN)

SVN является одной из клиент-серверных систем контроля версий и состоит из двух частей:

- Svnserve — серверная часть. *В качестве серверной части можно использовать обычный http-сервер.*
- SVN — клиентская часть

Можно создать и настроить собственный SVN-сервер или использовать готовые бесплатные сервисы: code.google.com , sourceforge.net, assembla.com.

Команды SVN

Перед тем как приступить к использованию SVN, ознакомимся с некоторыми командами, которые может выпонять клиентская часть.

Наиболее часто используемые разработчиками команды:

- **svn update** — обновляет содержимое локальной копии до самой последней версии из репозитория.
- **svn commit** — отправляет все изменения локальной копии в репозиторий.
- **svn add <*файл/папка*>** — включить файл/папку в локальную копию проекта

Нередко также используются команды:

- **svn move <*файл/папка1*> <*папка2*>** — переместить файл/папку1 в папку2
- **svn copy <*файл/папка*> <*папка*>** — скопировать файл/папку1 в папку2
- **svn delete <*файл/папка*>** — удалить файл/папку из локальной копии проекта

Прочие полезные команды SVN:

- **svn list <*URL*>** — просмотр каталога репозитория
- **svn log <*файл*> --verbose** — история изменения файла по ревизиям
- **svn cat --revision <*номер_ревизии*> <*файл*>** — отображение содержимого файла из данной ревизии
- **svn diff --revision <*номер_ревизии1:номер_ревизии2*> <*файл*>** — отображение изменений файла между двумя ревизиями
- **svn status** — отображение изменений в локальной копии относительно репозитория

Жизненный цикл изменений в проектах использующих SVN

Допустим, что у нас уже есть настроенный SVN-сервер, и мы хотим перенести на него проект. У нас есть папка projectsvn, в которой лежит три файла:

1.cpp
2.cpp
Makefile

Добавляем новые файлы на сервер:

svn import svn://...../repo/projectsvn projectsnv

Здесь *svn://...../repo/projectsvn* - url-адрес проекта, *projectsvn*- название добавляемой папки.

Получаем локальную копию файлов проекта:

svn checkout svn://...../repo/project project/projectsvn

project/projectsvn - адрес, где будет располагаться локальная копия.

После выполнения этой команды получим следующую структуру файлов:

```
project
  projectsvn
    .svn
    1.cpp
    2.cpp
    Makefile
```

Локальную копию не следует добавлять в ту папку, из которой делали import, так как при совпадении имён файлов они не будут перезаписаны.

Изменяем файлы

Допустим, на этом шаге мы хотим сделать какие-нибудь изменения в проекте, а именно:

- изменить содержимое 1.cpp следующим образом:

до изменений:	после изменений:

int main(){
  return 0;
}
	int main(){
	  printf(" ");
	  return 0;
	}

- удалить файл 2.cpp

Стоит обратить внимание на следующую особенность SVN: содержимое файлов проекта можно менять в любых привычных редакторах, **но** изменения в структуре файлов проекта следует выполнять с помощью соответствующих команд svn (add, copy, move, delete). То есть если мы просто удалим файл из папки с локальной копией - это никак не отразится на содержимом репозитория даже после успешного commit'a.

Фиксируем состояние

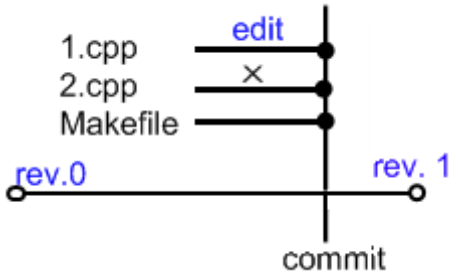
При фиксации происходит отправка всех изменений в локальной копии на сервер.

svn commit -m "....."

-m "....." — комментарий к commit'y.

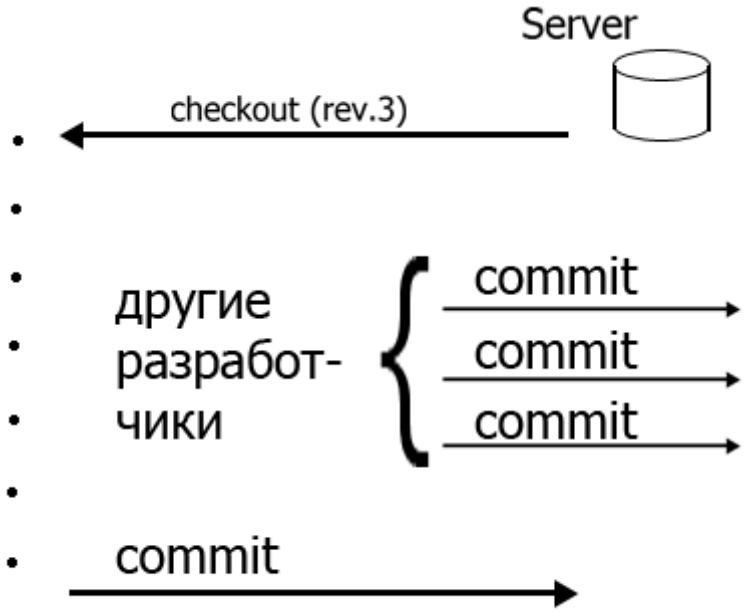
При выполнении этой команды SVN узнаёт нужную ему информацию о проекте из папки .svn
После каждого commit'a номер ревизии увеличивается на единицу.
Номер ревизии (Revision) &mdash целое число, показывающее номер состояния проекта.

Все выполненные действия можно схематически представить на рисунке:



Разрешение конфликтов

При работе над проектом возможны такие ситуации:



Например, в какой-то момент времени разработчик обновляется до последней версии проекта и начинает работу над некоторыми файлами. Через какое-то время, изменив файлы, он пытается сделать commit. Однако, этот commit не удастся, если другие разработчики уже вносили изменения в те же файлы проекта и сохраняли их в репозитории. Такая ситуация называется конфликтом. У разработчика, чей commit не

удался из-за конфликтов, есть два возможных варианта действий:

1. **svn revert** — отменить все свои изменения
2. **svn update** — забрать из репозитория новые версии файлов и, разрешив конфликты, снова попытаться сделать commit.

Как разрешаются конфликты

Допустим, в репозитории хранится файл **main.cpp** ревизии 13:

```
int main(){
  fprintf();
  return 0;
}
```

В какой-то момент времени user1 и user2 одновременно делают update:	svn update	svn update

Оба разработчика вносят изменения в main.cpp	int main(){
	  fscanf();
	  return 0;
	}
		int main(){
		  fpintfpr();
		  return 0;
		}

Затем user2 commit'ит свои изменения		svn commit
commit user1 не удастся, так как есть конфликт	svn commit	
Поэтому ему приходится делать update до последней версии	svn update	

Теперь у user1 есть несколько путей разрешения конфликта:

- убрать чужие изменения, оставив свои
- не вносить свои изменения, оставив всё как есть
- вручную соединить все изменения в одном файле

В последнем случае user1 имеет несколько файлов:

main.cpp — этот файл уйдёт в репозиторий после разрешений конфликта
main.cpp.mine — в этом файле хранятся изменения, сделанные user1
main.cpp.r13 — начальная версия файла без всяких изменений
main.cpp.r14 — файл, попавший в репозиторий (с изменениями user2)

После сведения всех изменений в **main.cpp** user1 пишет
svn resolved main.cpp (при этом удалятся временные файлы)
svn commit

SVN позволяет так же объединить любую версию с любым набором других версий с помощью update и merge.
Про команду merge можно прочитать, набрав в консоли команду svn help merge.

Задание

1. Зарегистрировать аккаунт на сайте assembla.com
2. Установить SVN на свой личный компьютер
3. Создать проект и сохранить изменения в систему контроля версий (в качестве проекта можно использовать любую лабораторную работу)
4. Разрешить доступ на изменение проекта другому пользователю(из Вашей группы)
5. Показать историю изменений проекта включая изменения внесенные другим пользователем.

Критерии приема лабораторной работы:

1. Наличие аккаунта и сохраненного проекта на сайте assembla.com
2. Наличие истории изменений проекта (сдающий л.р + 1 дополнительный пользователь)

Лабораторная работа №2

Распределенная система контроля версий “Git”

Краткая история Git

Как и многие замечательные вещи, Git начинался с, в некотором роде, разрушения во имя созидания и жарких споров. Ядро Linux — действительно очень большой открытый проект. Большую часть существования ядра Linux (1991-2002) изменения вносились в код путем приёма патчей и архивирования версий. В 2002 году проект перешёл на проприетарную РСУВ BitKeeper.

В 2005 году отношения между сообществом разработчиков ядра Linux и компанией разрабатывавшей BitKeeper испортились, и право бесплатного пользования продуктом было отменено. Это подтолкнуло разработчиков Linux (и в частности Линуса Торвальдса, создателя Linux) разработать собственную систему, основываясь на опыте, полученном за время использования BitKeeper. Основные требования к новой системе были следующими:

- Скорость
- Простота дизайна

- Поддержка нелинейной разработки (тысячи параллельных веток)
- Полная распределенность
- Возможность эффективной работы с такими большими проектами как ядро Linux (как по скорости, так и по размеру данных)

С момента рождения в 2005 г. Git разрабатывали так, чтобы он был простым в использовании, сохранив свои первоначальные свойства. Он невероятно быстр, очень эффективен для больших проектов, а также обладает превосходной системой ветвления для нелинейной разработки.

Основы Git

Главное отличие Git от любых других СУВ (например, Subversion и ей подобных) это то, как Git смотрит на данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору файлов и изменений сделанных для каждого из этих файлов во времени, как показано на рисунке 1.

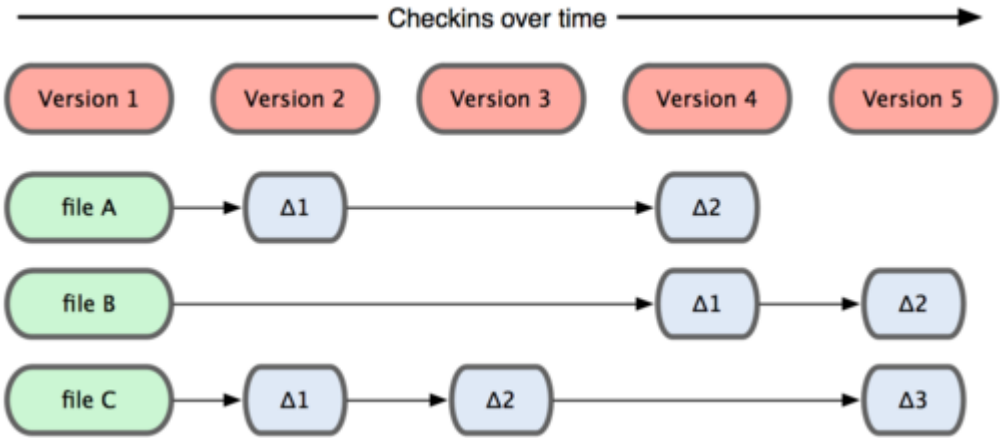


Рисунок 1. Другие системы хранят данные как изменения к базовой версии для каждого файла.

Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных, похоже на рисунок 2.

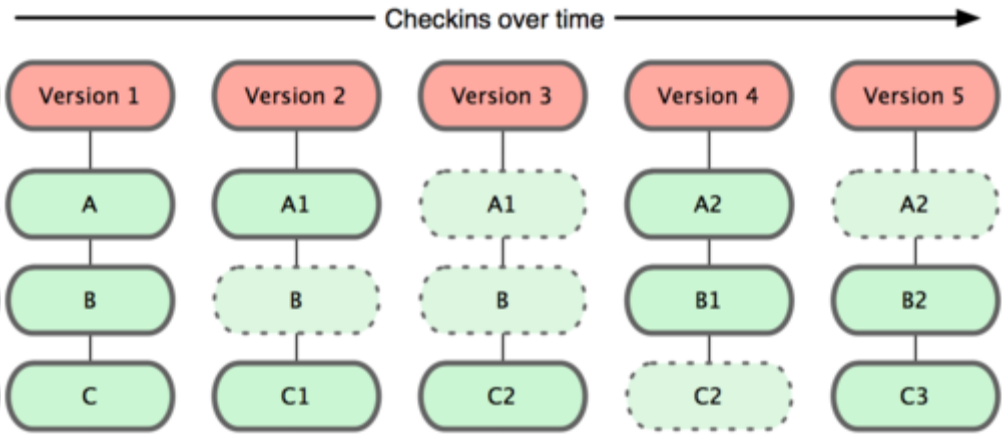


Рисунок 2. Git хранит данные как слепки состояний проекта во времени.

Это важное отличие Git от практически всех других систем управления версиями. Из-за него Git вынужден пересмотреть практически все аспекты управления версиями, которые другие системы взяли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто систему управления версиями.

Для совершения большинства операций в Git необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Если вы пользовались централизованными системами, где практически на каждую операцию накладывается сетевая задержка, вы, возможно, подумаете, что боги наделили Git неземной силой. Поскольку вся история проекта хранится локально у вас на диске, большинство операций выглядят практически мгновенными.

К примеру, чтобы показать историю проекта, Git-у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у сервера системы управления версиями или качать с него старую версию файла и делать локальное сравнение.

Git следит за целостностью данных

Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм, используемый Git для вычисления контрольных сумм, называется SHA-1 хеш. Это строка из 40 шестнадцатеричных знаков (0-9 и a-f), которая вычисляется на основе содержимого файла или структуры каталога, хранимого Git. SHA-1 хеш выглядит примерно так:

24b9da6552252987aa493b52f8696cd6d3b00373

Работая с Git, вы будете постоянно встречать эти хеши, поскольку они широко используются. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого.

Три состояния

Самое важное, что нужно помнить про Git, если вы хотите, чтобы дальше изучение шло гладко. В Git файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы – это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проекте с использованием Git есть три части: каталог Git (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).

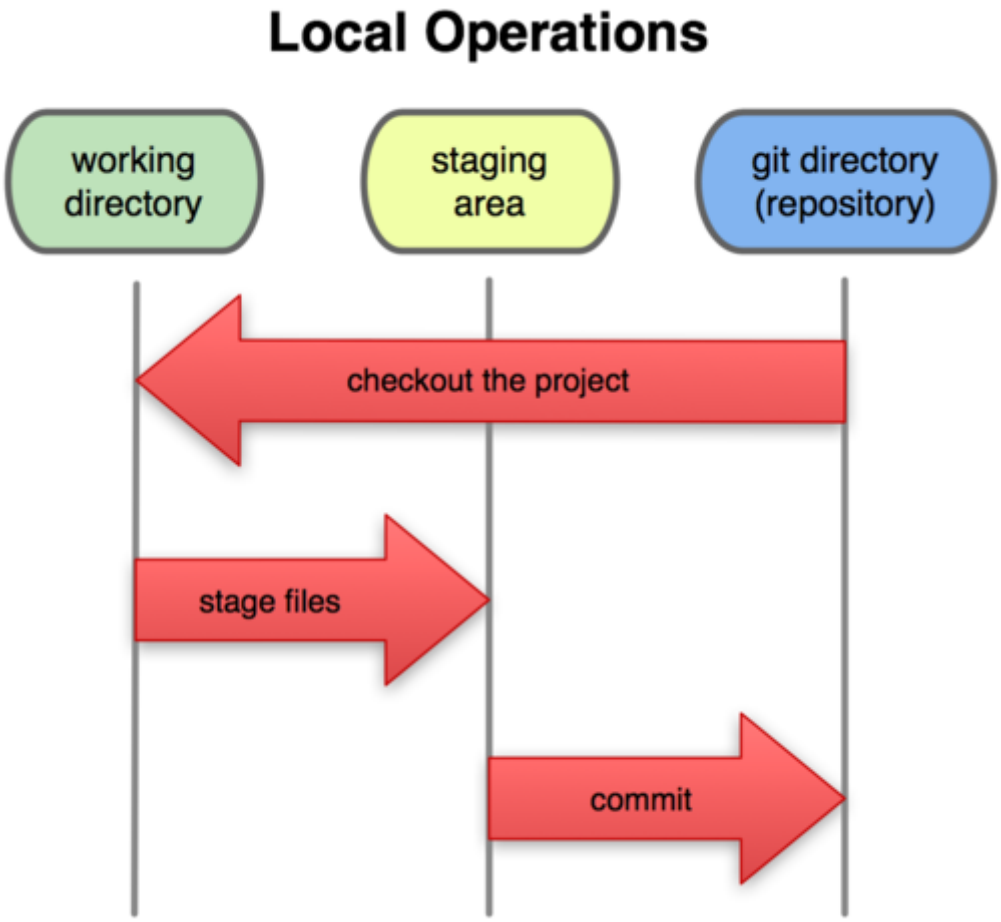


Рисунок 3. Рабочий каталог, область подготовленных файлов, каталог Git.

Каталог Git – это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Стандартный рабочий процесс с использованием Git выглядит примерно так:

1. Вы изменяете файлы в вашем рабочем каталоге.
2. Вы подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Вы делаете коммит. При этом слепки из области подготовленных файлов сохраняются в каталог Git.

Если рабочая версия файла совпадает с версией в каталоге Git, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

Больше о Git: http://git-scm.com/book/ru

Задание

1. Зарегистрировать аккаунт на сайте github.com
2. Установить git на свой личный компьютер
3. Создать проект и сохранить изменения в систему контроля версий (в качестве проекта можно использовать любую лабораторную работу)
4. Разрешить доступ на изменение проекта другому пользователю(из Вашей группы)
5. Показать историю изменений проекта включая изменения внесенные другим пользователем на сайте github.

Критерии приема лабораторной работы:

1. Наличие аккаунта и сохраненного проекта на сайте github.com
2. Наличие истории изменений проекта (сдающий л.р + 1 дополнительный пользователь)

Лабораторная работа №3

Pivotal Tracker - как средство сбора приоритезации и отслеживания выполнения задач в Scrum

Pivotal Tracker позволяет, в соответствие с методикой scrum, собирать, приоритезировать и отслеживать выполнение конкретных задач (story) - на разработку (feature), исправление ошибок (bug), вспомогательные работы (chore). Дополнительно и явно отслеживать планируемые релизы продукта (release).

Поле для интерпретации понятия "feature" довольно широкое, но согласно документации и рекомендациям методики - это "улучшения или новая функциональность, несущая непосредственную ценность для заказчика". Собственно, исходя из этого, принято учитывать в качестве "feature" - единицу самостоятельной функциональности проекта, которую пользователь может отдельно протестировать и использовать ("deliverable").

Планирование работы команды проводится на заранее определенном временном отрезке - этапе ("sprint"). Длительность этапа выбирается таким образом, что любая отдельно взятая функциональность ("feature") может быть реализована (значит - продумана, запрограммирована, оттестирована и принята заказчиком) в течение этого этапа. Эмпирически многие комманды приходят к двухнедельному спринту при времени на программирование одной истории - в среднем 2 дня.

Относительная сложность реализации каждой "feature" заранее оценивается в условных целочисленных баллах ("points") по фиксированной, заранее определенной шкале. Веса, назначенные задачам, должны больше отражать относительную (по отношению к другим задачам), нежели абсолютную (в человеко/часах, и пр). Таким образом, например, реализация "трехбалльной" истории по трудозатратам должна примерно совпадать с тремя "однобальными".

Сразу на выбор предлагается - линейная 1..3, степенная 2n, ряд Фибоначи. Можно ввести и свою шкалу, однако существующих шкал для проектов по разработке хватает, и, если подумать, предложенные шкалы имеют свой глубокий смысл.

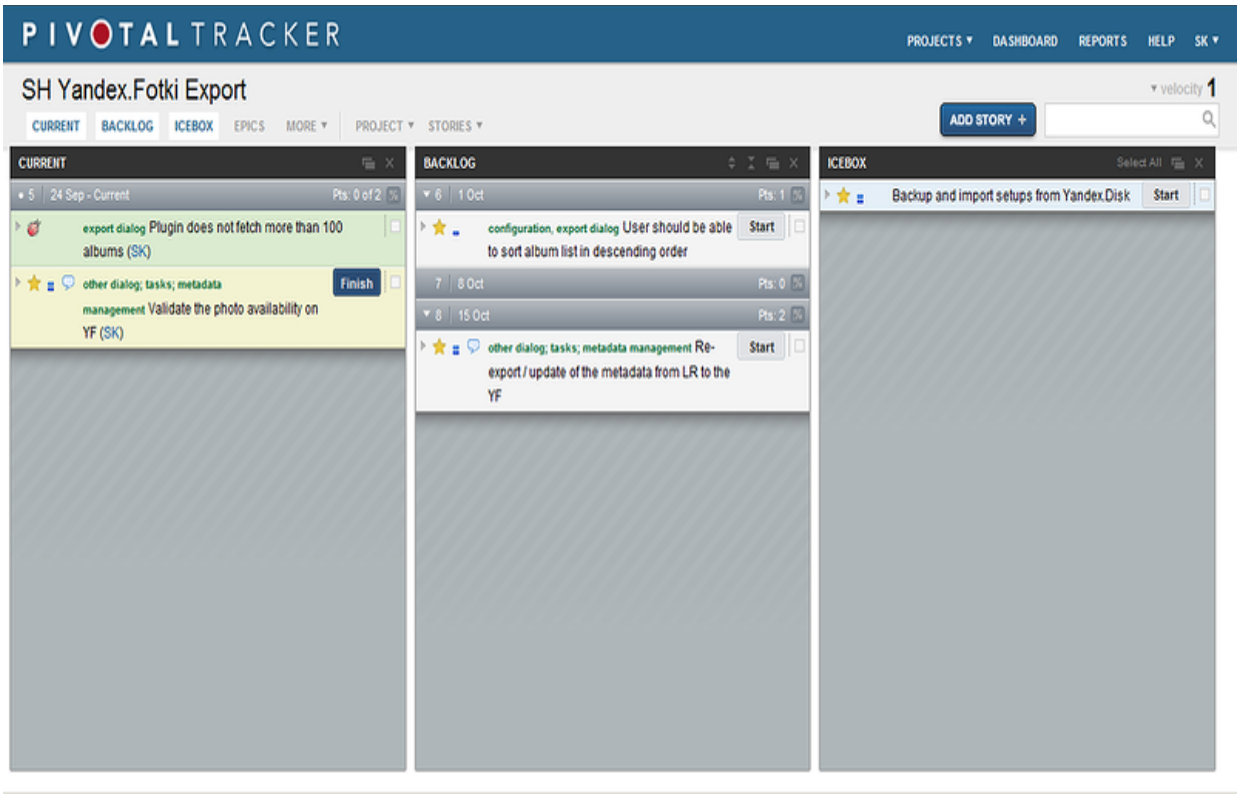
Оценке подлежат только истории, "полезные" для заказчика ("features") - ни исправления ошибок, ни вспомогательные работы оценке не подлежат.

Исходя из проведенной оценки историй, выбранной длительности этапа, и реальной статистики обработки списка историй, система вырабатывает значение важной проектной метрики - "скорость команды", количество реализованных балов за этап [points/sprint].

Шкала выбирается один раз на проект, и не меняется. Таким образом, используя метрику скорости можно осуществлять оперативное планирование и корректировку усилий в проекте.

Интерфейс пользователя и workflow

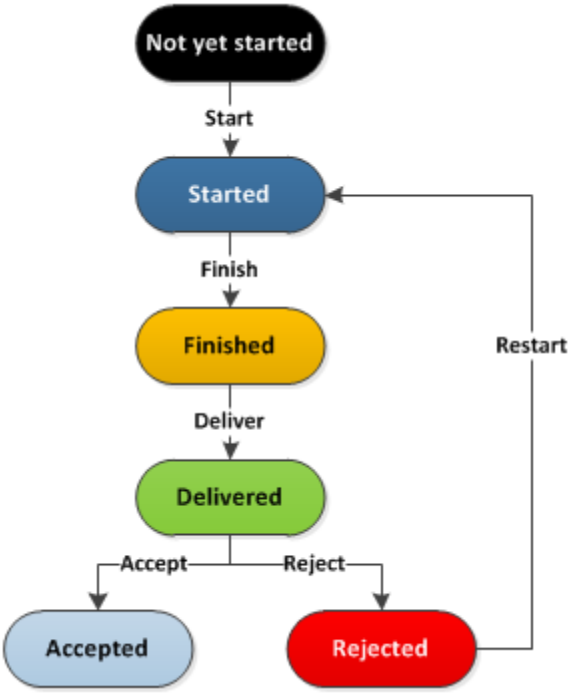
Все истории сразу после создания находятся в «замороженном состоянии», и пребывают там до тех пор, пока их не начнут исполнять, или они будут перемещены в список активных задач (current, backlog).



Каждая задача типа «feature», «bug» может находится в одном из следующих состояний:

1. Не начата
2. Начата
3. Закончена
4. Доставлена
5. Принята
6. Отвергнута

По статусам можно перемещаться свободно, но стандартный процесс выглядит так:

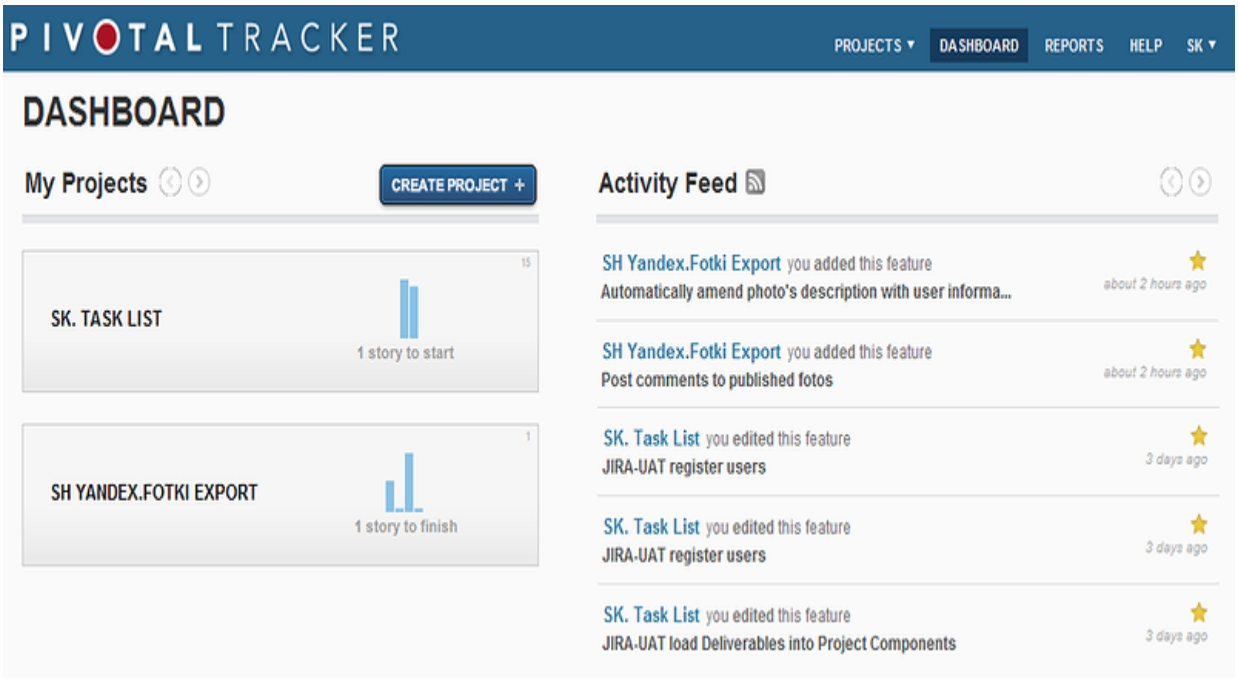


На строке каждой задачи появляется кнопка (или кнопки) соответствующая текущему состоянию задачи.

Отчеты и выгрузка данных

Project dashboard

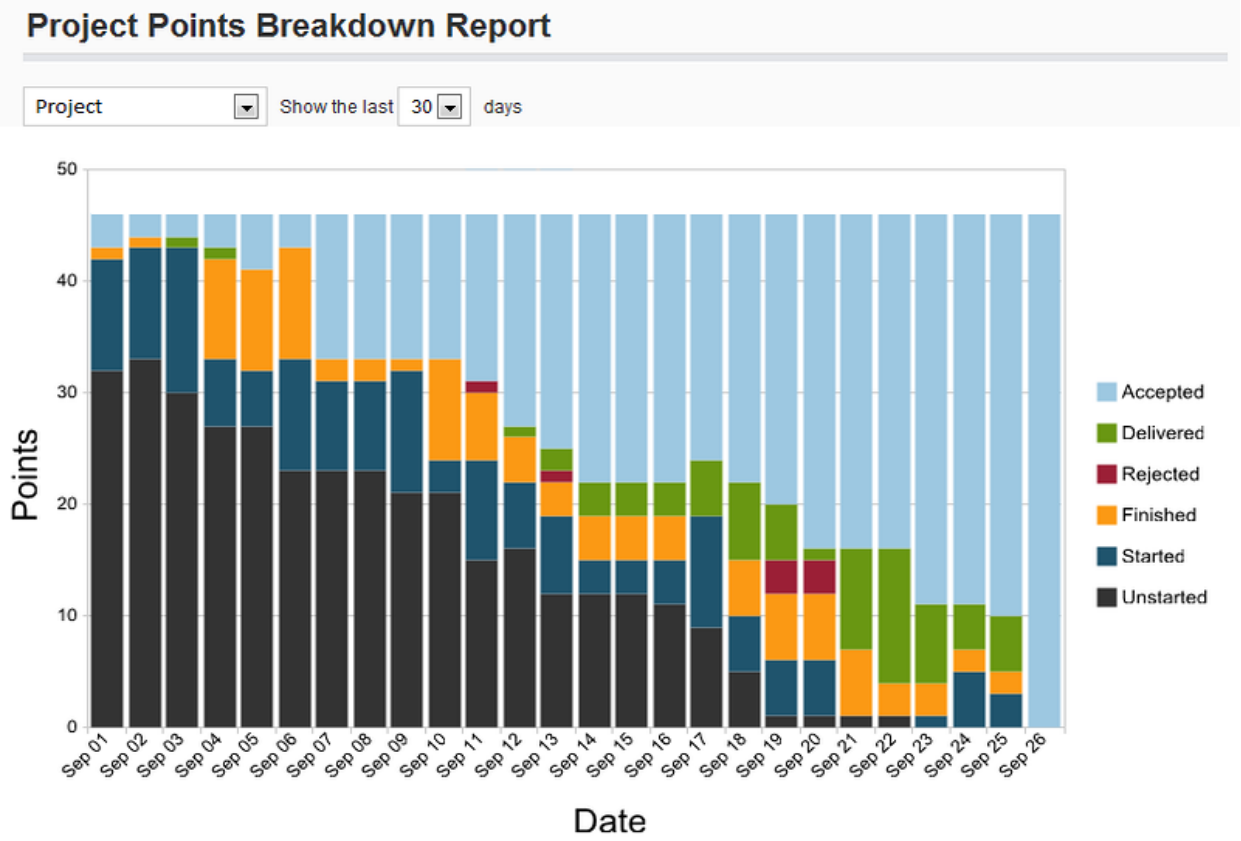
Первое что пользователь видит - это обзор состояния доступных ему проектов и ленту последних обновлений.



Для каждого проекта приводится миниатюрный график произвоительности, скорость проекта и сводка - что надо сейчас в проекте надо делать.

Points breakdown

Отчет "points breakdown" показывает распределение баллов в проекте по статусам, за определенный период времени.



Таким образом можно наблюдать проект в динамике.

Выгрузка данных в CSV формате

При выгрузке в CSV передается очень подробный отчет о статусе работы. В выгрузку можно включить завершенные, текущие и "замороженные" задачи. Понятно, что выгруженные данные можно использовать для любых целей - от анализа, до интеграйции.

Задание

1. Разбиться на комманды по 3-4 человека
2. Каждому члену команды созать аккаунт в Pivotal Tracker’е
3. С использованием Pivotal Tracker создать и распределить между членами команды задачи которые необходимо выполнить для разработки игры “БАЛДА” (правила игры: http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%BB%D0%B4%D0%B0_(%D0%B8%D0%B3%D1%80%D0%B0)
4. Оценить полученные задачи в очках (“points”)
5. Разбить задачи на два этапа (“sprints”) длительностью в две недели

Лабораторная работа №4

Компьютерная игра “БАЛДА”

Задание

На основании историй разработанных в лаброраторной работе №3 реализовать компьютерную игру “БАЛДА”. Каждый член команды защищает свои реализованные истории.