Методы организации работы в команде разработчиков. Системы контроля версий

Организация эффективной работы в команде

1. Цели и задачи:

- Первый шаг к эффективной работе - четкое определение целей и задач проекта. Команда должна понимать, какую проблему решает разрабатываемый продукт и какие функции он должен выполнять. Это позволит избежать недопониманий и сбоев на поздних этапах работы.

Организация эффективной работы в команде

2. Роли и обязанности:

- Каждый член команды должен знать свои роли и обязанности. Разработчики, дизайнеры, тестировщики - все должны понимать, какой вклад они вносят в проект. Четкое распределение ролей упрощает координацию и избегает дублирования усилий.

Организация эффективной работы в команде

3. Коммуникация:

- Открытая и эффективная коммуникация - ключевой аспект успешной работы в команде. Используйте средства коммуникации: персональные встречи, онлайн-чаты, видеоконференции. Регулярные обсуждения позволяют выявлять проблемы на ранних этапах и находить решения в сотрудничестве.

Организация эффективной работы в команде

4. Гибкость и адаптация:

- Развитие технологий и изменение требований рынка заставляют команду быть гибкой и адаптироваться к новым условиям. Методологии Agile (Scrum, Kanban и др.) помогают поддерживать гибкий подход к разработке, разбивая проект на короткие итерации.

Организация эффективной работы в команде

5. Управление задачами:

- Используйте инструменты для управления задачами, такие как доски Kanban или системы учета задач (например, Jira). Это поможет отслеживать ход выполнения, управлять приоритетами и контролировать сроки.

Организация эффективной работы в команде

6. Кодирование и ревью:

- Стандарты кодирования и регулярные код-ревью помогают обеспечивать качество кода и избегать ошибок. Обратная связь в рамках ревью способствует обучению и повышению навыков всей команды.

Организация эффективной работы в команде

7. Работа с ошибками:

- Ошибки - неизбежная часть разработки. Важно создать атмосферу, в которой члены команды не боятся допустить ошибку, но готовы извлекать уроки и предпринимать меры для их устранения.

Организация эффективной работы в команде

8. Мотивация и признание:

- Помните, что команда состоит из людей. Поддерживайте мотивацию, признавайте достижения и цените вклад каждого участника. Это создает позитивную рабочую атмосферу и способствует продуктивности.

Организация эффективной работы в команде

9. Обучение и саморазвитие:

- Технологии не стоят на месте, и разработчики также должны развиваться. Организуйте время для обучения новым технологиям, участвуйте в конференциях и воркшопах. Это поможет команде оставаться в тонусе и следовать современным практикам.

Системы контроля версий (Version Control System, VCS)

- Программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Зачем нужен контроль версий?

- Удобная работа в команде
- Отслеживание ошибок
- Возврат к ранней версии

Архитектуры VCS

- Локальная система контроля версий
- Централизованная система контроля версий
- Распределенная система контроля версий

Локальная система контроля версий (RCS)

Локальная система контроля версий (RCS)

Преимущества:
1. Позволяет хранить историю изменения файлов локально, без интернета.
2. Вы независимы от сторонних серверов.

Недостатки:
1. Вы можете потерять все файлы, если с вашем компьютером что-то случится.
2. Вы не можете работать в команде, поскольку репозиторий доступен только вам.

Централизованная система контроля версий (CVS, Subversion, Perforce)

Централизованная система контроля версий (CVS, Subversion, Perforce)

Преимущества:
1. Вы можете работать в команде с другими разработчиками.
2. Ваше начальство видит, чем вы занимаетесь.
3. У администратора есть четкий контроль, кто и что может делать.
Администрировать центральную VCS намного проще, чем локальные на каждой машине.

Недостатки:
1. Все данные хранятся только на одном сервере.
2. Если с сервером что-то случится, а копий данных нет, то весь проект может быть потерян.
3. Для работы необходим хороший интернет на протяжении целого дня.

Распределенная система контроля версий (Git, Mercurial, Bazaar)

Распределенная система контроля версий (Git, Mercurial, Bazaar)

Преимущества:
1. Работа компании теперь не зависит от работы сервера. Если сервер отключится, то каждый сотрудник продолжит работу с локальной копией репозитория, а после загрузит ее на сервер.
2. Можно работать с несколькими удаленными репозиториями, делиться кодом с другими людьми и коллаборировать целыми компаниями.

История Git

Git был разработан командой **Линуса Торвальдса** в 2005 году, как **open-source** аналог уже существующим системам. Но разработка Git не была спонтанным решением. Дело в том, что с самого первого релиза в 1991 году разработка ядра **Linux** выполнялась по старинке: старая версия архивировалась, а новые патчи от разработчиков становились новой версией.

Но с ростом популярности рос и объем данных, поэтому в 2002 году было принято решение перевести ядро **Linux** на распределенную систему управления версиями **BitKeeper** от **BitMover Inc**. Однако между компаниями произошел разлад и **BitMover Inc**. отозвали лицензию на бесплатное использование своего ПО.

Этот инцидент и подстегнул Линуса Торвальдса с командой разработчиков создать свою открытую распределенную систему контроля версий. Ребята хотели разработать надежное решение, обладающее высокой скоростью работы и упрощающее командную разработку.

Установка Git

- Для того, чтобы работать с Git, вам нужно поставить его на компьютер. Менеджер установки Git предлагает множество настроек. Большинство из них нужно оставить так, как они есть, но не всегда. Существует множество руководств по установке Git

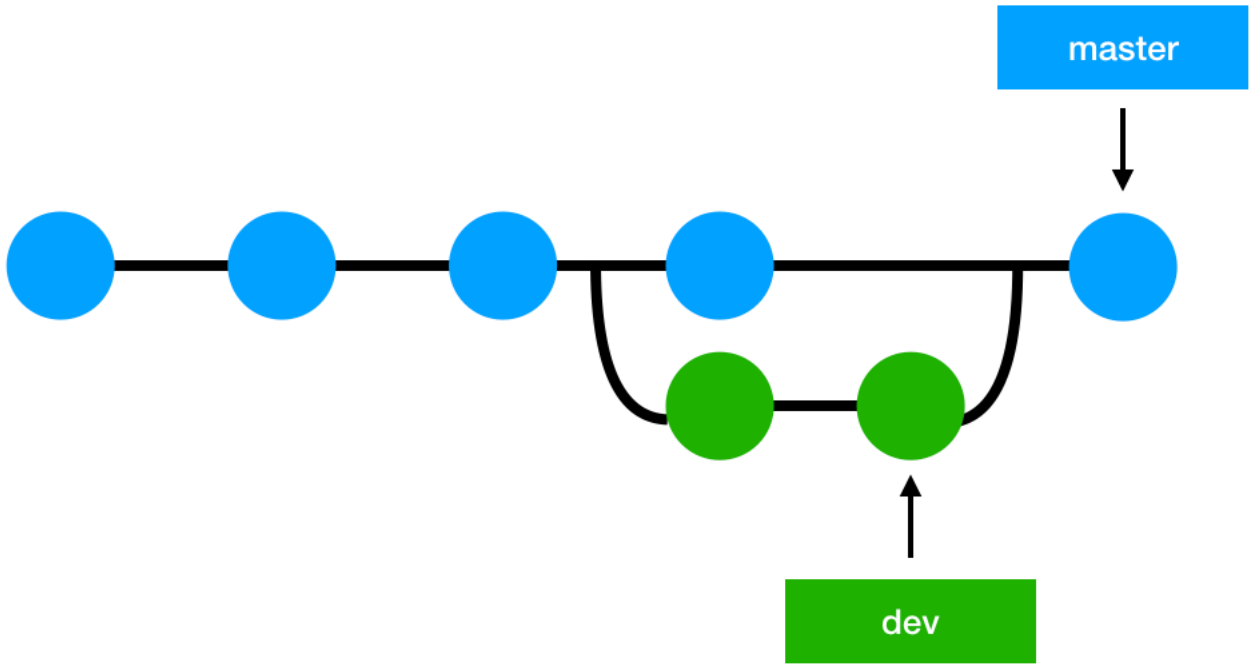
Основные понятия

- **Репозиторий** – папка проекта, отслеживаемого Git, содержащая дерево изменений проекта в хронологическом порядке. Все файлы истории хранятся в специальной папке **.git/** внутри папки проекта.
- **Индекс** – файл, в котором содержатся изменения, подготовленные для добавления в коммит. Вы можете добавлять и убирать файлы из индекса.
- **Коммит** – фиксация изменений, внесенных в индекс. Другими словами, коммит – это единица изменений в вашем проекте. Коммит хранит измененные файлы, имя автора коммита и время, в которое был сделан коммит. Кроме того, каждый коммит имеет уникальный идентификатор, который позволяет в любое время к нему откатиться.
- **Указатели HEAD**, **ORIGHEAD** и т. д. – это ссылка на определенный коммит. Ссылка – это некоторая метка, которую использует Git или сам пользователь, чтобы указать на коммит.
- **Ветка** – это последовательность коммитов. Технически же, ветка – это ссылка на последний коммит в этой ветке. Преимущество веток в их независимости. Вы можете вносить изменения в файлы на одной ветке, например, пробовать новую функцию, и они никак не скажутся на файлах в другой ветке. Изначально в репозитории одна ветка, но позже мы рассмотрим, как создавать другие.
- **Рабочая копия**. Директория **.git/** с её содержимым относится к Git. Все остальные файлы называются рабочей копией и принадлежат пользователю

Команды

- **git init** – создает новый репозиторий
- **git status** – отображает список измененных, добавленных и удаленных файлов
- **git add** – добавляет указанные файлы в индекс
- **git commit** – фиксирует добавленные в индекс изменения
- **git clone** – клонирование удаленного репозитория
- **git fetch** – получение изменений
- **git push** – отправка изменений

Ветки



Зачем нужны ветки?

- Ветки нужны, чтобы несколько программистов могли **вести работу** над одним и тем же проектом или даже файлом **одновременно**, при этом не мешая друг другу.
- Кроме того, ветки используются **для тестирования экспериментальных функций**: чтобы не повредить основному проекту, создается новая ветка специально для экспериментов.
- Помимо прочего, ветки можно использовать **для** разных выходящих **параллельно релизов** одного проекта. Например, в репозитории Python может быть две ветки: **python-2** и **python-3**.

Команды

- **git branch** – создание новой ветки
- **git checkout** – переключение на другую ветку
- **git merge** – слияние веток

Конфликты слияния

- Очень часто во время слияния веток оказывается, что ваши изменения удаляют или переписывают информацию в уже существующих файлах. Такая ситуация называется **файловым конфликтом**. Git останавливает выполнение слияния, пока вы не разрешите конфликт.