Разработка системы контроля версий

Необходимо разработать клиент-серверное приложение, которое должно обеспечить поддержку версионности файлов в каталоге клиента, а также набор команд для управления данной системой.

Серверный компонент должен обрабатывать все входящие команды от клиента, а так же работать с разными репозиториями разных клиентов. Это значит, что может быть как несколько клиентов с каталогами, так и один клиент с разными каталогами, для которых необходима версионность. Сервер должен содержать следующее:

1. Всю необходимую функциональность для работы с сетью. Должно быть инкапсулировано в логическую цепочку классов.
2. Диспетчер потоков для многопоточной обработки команд в пределах разных репозиториев.
3. Очередь выполнения команд в пределах одного репозитория(разобраться, есть ли необходимость).
4. Фабрику команд, которая преобразует запрос клиента в конкретную команду.
5. Унифицированный интерфесй IDataProvider, обеспечивающий доступ к данным конкретных репозиториев. Это необходимо для того, чтобы иметь возможность сменить источник данных, например, на базу данных без изменений в самой системе.

Требование: реализовать соответствующий класс FolderProvider, который должен обеспечить доступ к репозиториям, хранящимся в каталогах сервера. Репозиторий должен хранить инкрементальные версии и полные(подумать, как часто необходимо делать полные копии).

Клиентский компонент представляет собой командный интерфейс для управления данной системой. Клиент должен содержать:

1. Всю необходимую функциональность для работы с сетью. Должно быть инкапсулировано в логическую цепочку классов. Не должно пересекаться с серверным API, либо тогда использование серверного API. Необходимо избежать дублирования кода.
2. Поддержку ввода команд с консоли.
3. Поддержку вывода прогресса выполнения команды(процент архивирования, процент передачи и т.д.)

4. Вывод результатов выполнения команды.

Обмен между клиентом и сервером осуществляется в формате пересылки "пакетов", которые интерпретируются на каждой стороне. Пакет представляет собой унифицированный интерфейс ICommandPacket, содержащий симметричные методы сериализации и десериализации. Т.е., необходимо обеспечить в любой момент смену формата пересылки пакетов без изменения основной функциональности самой системы. При выполнении команды необходимо посылать ответный пакет, который содержит результат выполнения, а так же:

1. int error - код ошибки, если успешно, то 0;
2. string errorInfo - текст ошибки, если успешно, то "OK"

Требование: все пересылаемые данные между клиентом и сервером должны быть архивированы.

Пожелание: обеспечить шифрование пересылаемых данных.

Клиент и сервер могут содержать "общий" код. Это значит, что такой "общий" код должен быть вынесен в отдельные классы-утилиты, доступные каждому проекту. Например, интерфейс пакетов.

Механизм наращивания версии должен быть выделен в отдельные классы и интерфейсы, чтобы была возможность, например, механизм, который наращивает версии по принципу 1.0 -> 1.1 -> 1.2, заменить на механизм по типу 1.01->1.02 и т.д. или любой другой. Это актуально только до создания конкретного репозитория. Для уже созданных репозиториев изменить механизм наращивания версии уже невозможно.

Список необходимых команд

1. Команда **ADD**. Выполняет создание пустого репозитория на сервере. Сигнатура: **add repoName**, где add - ключевое слово, repoName - уникальное имя репозитория на сервере.

2. Команда **CLONE**. Выполняет "клонирование" содержимого репозитория сервера в локальный каталог клиента. Если каталог не пуст, то необходимо его очистить. Данная команда устанавливает связь между локальным каталогом на клиенте и существующим репозиторием на сервере.

Сигнатура: **clone path repoName flags**, где clone - ключевое слово, path - путь к локальному каталогу, repoName - имя репозитория на сервере, flags - флаги управления.

Требование: реализовать поддержку флага "точка". Т. е. если после имени репозитория через пробел стоит "."(точка), то необходимо скопировать содержимое репозитория в папку строго по пути параметра path. Если этого флага нет, то в каталоге path необходимо создать папку с именем repoName, а затем в него скопировать содержимое репозитория.

Все последующие команды применимы только в том случае, если первой командой была clone. Иными словами, команды работают только с репозиторием, который был клонирован на клиент.

3. Команда **UPDATE**. Выполняет обновление каталога клиента до последней версии, которая есть на сервере. Сигнатура: **update**, где update - ключевое слово.

4. Команда **COMMIT**. Фиксирует изменения в репозиторий и наращивает версию. Сигнатура: **commit**, где commit - ключевое слово. При выполнении данной команды происходит отправка на сервер тех файлов, что были изменены относительно текущей версии. Пример:

   1.txt
   2.txt
   3.txt – файлы лежат локально в каталоге клиента.
   Выполняем commit – все три файла уходят на сервер, и в репозитории создается каталог 1.0(к примеру) –первая версия, внутри каталога лежат все три файла.

   Затем, изменим каким-либо образом файл 3.txt.
   Выполняем commit – на сервер отправляется только файл 3.txt, создается каталог 1.1, в котором и будет хранится этот файл. И так

далее, т. е. система хранит только измененные файлы относительно первой версии.

Требование: реализовать возможность удаления: удаляя файл из каталога клиента и выполняя команду commit, сервер должен нарастить версию и понять, что с этой версии файл считается удаленным. Но возможен возврат на какую-либо версию, в которой удаленный файл еще существовал(команда **REVERT**) .

5. Команда **REVERT**. Выполняет откат на заданную версию. Сигнатура: **revert version flags**, где revert - ключевое слово, version - версия, до которой надо откатиться. flags - флаги выполенения.

   Требование: реализовать поддержку флага "-hard". Если такой флаг стоит, то значит необходимо полностью восстановить копию версии с сервера, в том числе измененные файлы. Если флага нет, необходимо восстановить только те файлы, которые соответствуют версии, с которой откатываемся. Файлы, которые модифицированы уже не в рамках откатываемой версии, измениться не должны.

   Требование: если version отсутствует, то необходимо откатиться с использование флага "-hard" до текущей версии репозитория. Т. е. результат выполения команды в данном случае совпадает с результатом команды clone - получение последней версии и удаление всего лишнего.

6. Команда **LOG**. Команда должна вывести историю коммитов в форматированном виде. Дату, ip-адрес, версию коммита, список закомиченных файлов: добавленных и удаленных. Сигнатура: **log**, где log - ключевое слово.

Общие замечания

При выполнении поставленной задачи необходимо исходить из основных принципов ООП. Идеально работающая задача без соответствующей структуры исходных кодов не будет зачтена **вообще**. Проектирование архитектуры является неотъемлемой частью разработки. Вам предоставляется свобода в проектировании подробной архитектуры классов для данной задачи. Если Вы не сможете аргументировать ту или иную структуру данных, то это равносильно описанному выше - отсутствию структуры. Также необходимо предусмотреть случаи, которые не описаны в данной постановке, например, все команды должны выполняться только, если была команда clone. Иными словами, если есть просто каталог, то никакой версионности еще нет. Версионность появляется при ассоциации с серверным репозиторием, что происходит по команде clone. Итогом выполнения данной задачи является создание простой стабильной гибкой системы контроля версий, расширяемой путем добавления новых команд.

Приложения

1. Необходимо отделять код бизнес-логики от кода, обеспечивающего инфраструктуру. В обработчике, принимающем запрос от клиента, должен возникнуть некий общий код:

ICommandPacket packet = ClienUtils.getRequest(client) – здесь происходит десериализация запроса от клиента в пакет данных.

ICommand command = CommandFactory.createInstance(packet) – здесь создаем конкретную команду на выполнение.

ICommandPacket response = command.execute() – выполняем команду.

ClientUtils.sendResponse(response) - сериализуем и отправляем ответ клиенту.

Приведенный код является псевдокодом, показывающим главную особенность: **расширение функциональности системы должно быть горизонтальным – путем добавления новых команд без изменения обрамляющей логики. Весь общий код должен быть инкапсулирован в соответствующих классах.**

Тестирование

Во время тестирования будут производиться коммиты в репозиторий, переключение версий, сравнение содержимого на сервере и на клиенте. Необходимо обеспечить вывод монитора потоков, который должен отражать в реальном времени появление и завершение потоков клиента.