**1 КОНЦЕПЦИЯ ПРОГРАММНОЙ СИСТЕМЫ**

* 1. **Проблемы разработки программного обеспечения**

Наиболее распространёнными проблемами, возникающими в процессе разработки программного обеспечения (ПО), считают:

* недостаток прозрачности. В любой момент времени сложно сказать, в каком состоянии находится проект и каков процент его завершения;
* недостаток контроля. Без точной оценки процесса разработки срываются графики выполнения работ и превышаются установленные бюджеты;
* недостаток трассировки;
* недостаток мониторинга. Невозможность наблюдать ход развития проекта не позволяет контролировать ход разработки в реальном времени;
* неконтролируемые изменения. У потребителей постоянно возникают новые идеи относительно разрабатываемого программного обеспечения;
* недостаточная надежность. Самый сложный процесс – поиск и исправление ошибок в программах на ЭВМ;
* отсутствие гарантий качества и надежности программ из-за отсутствия гарантий отсутствия ошибок в программах вплоть до формальной сдачи программ заказчикам.

Несмотря на все эти проблемы, в отличие от человека, для компьютера не существует понятия качества программы, так как он оперирует только с машинным кодом. Но то, как написан код, может сильно влиять не только на процесс [сопровождения ПО](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BF%D1%80%D0%BE%D0%B2%D0%BE%D0%B6%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), но и на его создание. О качестве исходного кода можно судить по следующим параметрам:

* читаемость кода (в том числе наличие [комментариев](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%80%D0%B8%D0%B8_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) к коду);
* лёгкость в поддержке, тестировании, отладке и устранении ошибок, модификации и портировании;
* экономное использование ресурсов: памяти, процессора, дискового пространства;
* отсутствие замечаний, выводимых компилятором;
* отсутствие «мусора» — неиспользуемых переменных, недостижимых блоков кода, ненужных устаревших комментариев и т. д.;
* адекватная обработка ошибок;
* переносимость — возможность использования обработчика (компилятора, интерпретатора, транслятора) разных версий или даже различных [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0);
* возможность [интернационализации](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) интерфейса.

В программировании, в большинстве случаев, для определения качества кода достаточно знать две характеристики:

* соответствие правилам. Под этот пункт подпадают ситуации, когда код компилируется и работает. Но зачем нужны правила написания кода, если код работает? Чтобы ответить на вопрос выделим несколько типов правил:
* синтаксические правила — одни из наиболее спорных правил, поскольку совсем никоим образом не виляют на исполнение программы. К ним можно отнести стиль именования переменных (обычно camelCase или через подчеркивание), констант (обычно UPPERCASE), методов, стиль написания фигурных скобок и нужны ли они если в блоке только одна строка кода;
* правила поддержки кода — правила, которые должны сигнализировать что код слишком сложный и его будет трудно сопровождать. В качестве примера можно привести наличие дубликатов в коде или “магических цифр”;
* очистка и оптимизация кода — самые простые правила, так как обычно строки, которые не используются, не нужны. Сюда можно отнести подключения лишних библиотек, переменные и методы, которые уже не используются, но по какой-то причине их оставили;
* сложность кода. Цикломатическая сложность кода – характеристика, от которой напрямую зависит сложность поддержки кода. Этот параметр зависит от количества вложенных операторов ветвления и циклов. Чем индекс ниже, тем лучше, и тем легче в будущем будет менять структуру кода.

Использование возможностей систем контроля версий, а именно комментирование версий, помогает снизить сложность кода.

**1.2 Использование систем контроля версий программного обеспечения**

Ситуация, в которой электронный документ за время своего существования претерпевает ряд изменений, достаточно типична. При этом часто бывает важно иметь не только последнюю версию, но и несколько предыдущих. В простейшем случае можно просто хранить несколько вариантов документа, нумеруя их соответствующим образом. Такой способ неэффективен (приходится хранить несколько практически идентичных копий), требует повышенного внимания и дисциплины и часто ведёт к ошибкам, поэтому были разработаны средства для автоматизации этой работы.

Традиционные системы управления версиями используют централизованную модель, когда имеется единое хранилище документов, управляемое специальным сервером, который и выполняет бо́льшую часть функций по управлению версиями. Пользователь, работающий с документами, должен сначала получить нужную ему версию документа из хранилища; обычно создаётся локальная копия документа, так называемая «рабочая копия». Может быть получена последняя версия или любая из предыдущих, которая может быть выбрана по номеру версии или дате создания, иногда и по другим признакам. После того, как в документ внесены нужные изменения, новая версия помещается в хранилище. В отличие от простого сохранения файла, предыдущая версия не стирается, а тоже остаётся в хранилище и может быть оттуда получена в любое время. Сервер может использовать т. н. дельта-компрессию — такой способ хранения документов, при котором сохраняются только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных. Поскольку обычно наиболее востребованной является последняя версия файла, система может при сохранении новой версии сохранять её целиком, заменяя в хранилище последнюю ранее сохранённую версию на разницу между этой и последней версией. Некоторые системы (например, ClearCase) поддерживают сохранение версий обоих видов: большинство версий сохраняется в виде дельт, но периодически (по специальной команде администратора) выполняется сохранение версий всех файлов в полном виде; такой подход обеспечивает максимально полное восстановление истории в случае повреждения репозитория.

Иногда создание новой версии выполняется незаметно для пользователя (прозрачно), либо прикладной программой, имеющей встроенную поддержку такой функции, либо за счёт использования специальной файловой системы. В этом случае пользователь просто работает с файлом, как обычно, и при сохранении файла автоматически создаётся новая версия.

Часто бывает, что над одним проектом одновременно работают несколько человек. Если два человека изменяют один и тот же файл, то один из них может случайно отменить изменения, сделанные другим. Системы управления версиями отслеживают такие конфликты и предлагают средства их решения. Большинство систем может автоматически объединить (слить) изменения, сделанные разными разработчиками. Однако такое автоматическое объединение изменений, обычно, возможно только для текстовых файлов и при условии, что изменялись разные (непересекающиеся) части этого файла. Такое ограничение связано с тем, что большинство систем управления версиями ориентированы на поддержку процесса разработки программного обеспечения, а исходные коды программ хранятся в текстовых файлах. Если автоматическое объединение выполнить не удалось, система может предложить решить проблему вручную.

Часто выполнить слияние невозможно ни в автоматическом, ни в ручном режиме, например, если формат файла неизвестен или слишком сложен. Некоторые системы управления версиями дают возможность заблокировать файл в хранилище. Блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла (например, средствами файловой системы) и обеспечивает, таким образом, исключительный доступ только тому пользователю, который работает с документом.

Многие системы управления версиями предоставляют ряд других возможностей:

− позволяют создавать разные варианты одного документа, так называемой ветки, с общей историей изменений до точки ветвления и с разными — после неё;

− дают возможность узнать, кто и когда добавил или изменил конкретный набор строк в файле;

− ведут журнал изменений, в который пользователи могут записывать пояснения о том, что и почему они изменили в данной версии;

− контролируют права доступа пользователей, разрешая или запрещая чтение и/или изменение данных, в зависимости от того, кто запрашивает это действие.

**1.3 Жизненный цикл программного продукта**

При работе с системами контроля версий важно понимать, что для получения наибольшего эффекта внедрение тестирования должен реализовываться параллельно с жизненным циклом разработки системы. Методология проектирования информационных систем описывает процесс создания и сопровождения систем в виде жизненного цикла информационной системы, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполнения работ, ожидаемые результаты, роли участников, методы и средства, необходимые для выполнения работ и т.д. Такое формальное описание позволяет управлять данными процессами и организовать процесс коллективной разработки. Жизненный цикл информационной системы можно представить, как ряд событий, которые происходят с ней во время создания и использования.

Модель жизненного цикла описывает различные состояния системы, начиная с момента возникновения необходимости в данной системе и заканчивая моментом ее выхода из эксплуатации. Модель жизненного цикла – это структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы.

На настоящий момент основными моделями жизненного цикла являются следующие модели:

* каскадная модель;
* поэтапная модель с промежуточным контролем;
* спиральная модель.

Каскадная модель предполагает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем. Пример каскадной модели изображен на рисунке 1.1.

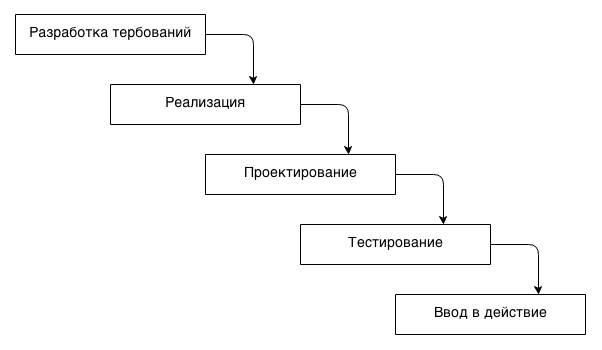


Рисунок 1.1 – Каскадная модель

Следуя каскадной модели, разработчик переходит от одной стадии к другой строго последовательно. Сначала полностью завершается этап «определение требований», в результате чего получается список требований к ПО. После того как требования полностью определены, происходит переход к проектированию, в ходе которого создаются документы, подробно описывающие для программистов способ и план реализации указанных требований. После того как проектирование полностью выполнено, программистами выполняется реализация полученного проекта. На следующей стадии процесса происходит интеграция отдельных компонентов, разрабатываемых различными командами программистов. После того как реализация и интеграция завершены, производится тестирование и отладка продукта; на этой стадии устраняются все недочёты, появившиеся на предыдущих стадиях разработки. После этого программный продукт внедряется и обеспечивается его поддержка — внесение новой функциональности и устранение ошибок.

Тем самым, каскадная модель подразумевает, что переход от одной фазы разработки к другой происходит только после полного и успешного завершения предыдущей фазы, и что переходов назад либо вперёд или перекрытия фаз — не происходит.

Пример поэтапной модели с промежуточным контролем изображен на рисунке 1.2.

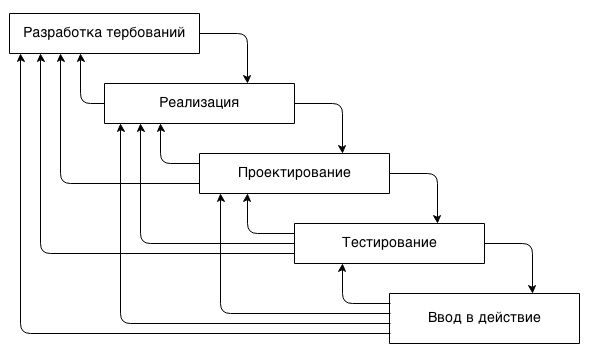


Рисунок 1.2 – Поэтапная модель с промежуточным контролем

При использовании поэтапной модели разработка ведется итеративно с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующие влияния результатов разработки друг на друга на различных этапах. Преимущество такой модели заключается в том, что межэтапные корректировки обеспечивают меньшую трудоемкость по сравнению с каскадной моделью; однако, время жизни каждого из этапов растягивается на весь период разработки.

В спиральной модели разработки жизненный цикл напоминает спираль – на каждом витке выполняется создание очередной версии продукта, конкретизируются и уточняются требования, определяется качество и планируются работы следующего витка. Особое внимание уделяется начальным этапам разработки – анализу и проектированию, где реализация тех или иных технических проверяется путем создания прототипов. Пример спиральной модели изображен на рисунке 1.3.

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

На практике наибольшее распространение получили каскадная и спиральная модель. Причем каскадная модель была более популярна в семидесятых, начале восьмидесятых годов, так как в то время в проектах достаточно простых информационных систем представляло собой единый, функционально и информационно независимый блок.

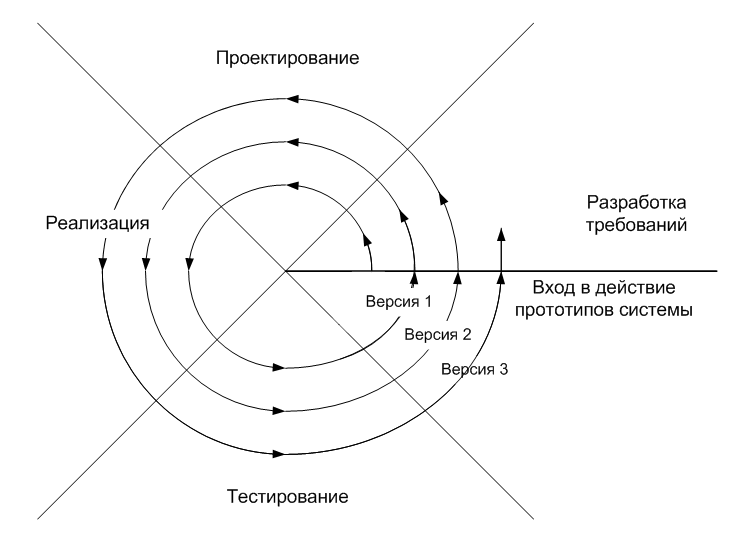


Рисунок 1.3 – Спиральная модель

Во всех вышеперечисленных моделях большую роль играют системы контроля версий. С их помощью можно добиться высокого уровня контроля на каждом шаге системы, как с точки зрения ведущего программиста – руководителя разработки, так и с точки зрения управляющего проектом.

Дополнительно ко всем преимуществам, описанным выше, некоторые дополнения к системам контроля версий поддерживают дополнительную функциональность, необходимую при управлении проектом.

Например, система JIRA (система отслеживания ошибок, предназначенная для организации общения с пользователями, хотя в некоторых случаях систему можно использовать для управления проектами) на данный момент используется подавляющим большинством крупных аутсорсинговых компаний. Кроме того, целый ряд компаний используют JIRA не только в качестве баг-трекера, но и как систему управления проектами.

Такую систему очень удобно использовать совместно с системой контроля версий, так как чаще всего ошибки, присланные пользователем, должны быть исправлены как можно быстрее и очень важно уметь правильно отслеживать её текущий статус. Именно поэтому крупные компании с большим штатом разработчиков требуют знания от своих сотрудников систем контроля версий, совместно с системами баг-трекинга.

Все системы контроля версий делятся на централизованные и децентрализованные. Соответственно, в централизованных системах контроля версий присутствует центральный сервер, на котором завязаны все остальные системы. В децентрализованных системах такой сервер тоже есть, но он только синхронизирует работу, так как, в принципе, копия версии с любого компьютера может стать основной.

Для проведения лабораторного практикума были выбраны обе системы контроля версий. На практике чаще всего используются децентрализованные, как более надёжные и простые в использовании. Тем не менее, именно с использованием централизованных систем контроля версий на данный момент написано наибольшее количество программных комплексов, так как они исторически являлись первыми.

**1.4 Концепция практикума по централизованным системам**

В разрабатываемом практикуме первой лабораторной было решено сделать работу с программным обеспечением, используя централизованную систему контроля версий. Исторически сложилось, что с приходом более совершенных IDE и языков программирования, вопрос взаимодействия с коллегами является ключевым для многих разработчиков. В основном чтобы решить эту проблему, начали появляться централизованные системы контроля версий (ЦСВ или CVS – Centralized Version Control systems). Примерами таких систем являются:

− CVS;

− Subversion;

− Perforce.

В них используется следующий метод сохранения информации: имеется один сервер, который хранит все версии файлов, а также некоторое количество клиентов, на которые поставляется информация с сервера. Диаграмма таких взаимоотношений представлена на рисунке 1.4.



Рисунок 1.4– Диаграмма взаимоотношений в централизованных системах контроля версий

Такой подход имеет много преимуществ, особенно относительно работы без систем контроля версий. Основное из них в том, что разработчики могут работать с одним и тем же файлом одновременно. Кроме этого, есть ещё и другие плюсы, например, каждый разработчик в любой момент времени может знать, что делают все остальные; кроме этого, администраторы систем, имея точно такую же возможность, могут эффективнее управлять своими кадрами и всем продуктом в целом.

Но, несмотря на решение достаточно большого спектра проблем, у таких систем есть свои недостатки. Наиболее существенный – единая точка отказа. Если сервер по каким-то причинам выходит из строя, то все работают только со своими локальными копиями, не имея возможности смотреть изменения других разработчиков.

В первой лабораторной работе учащимся будет предложено использовать СКВ Subversion, как наиболее популярную и хорошо документированную централизованную систему. Система Subversion, часто сокращаемая до SVN, из-за основной команды svn – система контроля версий, которая была выпущена для замены устаревшей CVS ещё в 2004 году. СКВ Subversion реализует все основные функции CVS и свободна от ряда недостатков последней.

Система Subversion, как централизованная система (в отличие от распределённых систем, таких как Git или Mercurial), то есть данные хранятся в едином хранилище. Хранилище может располагаться на локальном диске или на сетевом сервере. СКВ Subversion используется многими сообществами разработчиков открытого программного обеспечения (в том числе сообществами, ранее использовавшими CVS).

Чтобы лучше понять системы контроля версий, необходимо изучать их на примере реальной командной работы с программными продуктами. Поэтому в лабораторном практикуме мы также будем использовать скриптовые языки программирования, как не требующие много времени для изучения и позволяющие в одну строку написать функционал, позволяющий, например, вызвать модальное окно с каким-либо сообщением.

Для первой лабораторной работы было решено выбрать утилиту zenity. Утилита zenity – это средство создания диалоговых окон в режиме командной строки. Следует отметить, что на самом деле диалоговые окна создаются средствами Gtk+, поэтому в системе должны быть установлены соответствующие библиотеки.

Чтобы начать использовать утилиту zenity на практике не требуется обладать особыми знаниями или умениями, достаточно просто познакомиться с различными опциями (ключами), позволяющими в полной мере использовать возможности этой программы.

В данной утилите определены четыре типа диалоговых окон для вывода сообщений:

− ошибка (ключ --error);

− информация (ключ --info);

− вопрос (ключ --question);

− предупреждение (ключ --warning).

Для того, чтобы создать простейшее диалоговое окно, необходимо выполнить команду: zenity --entry --title='addition' --text='enter first number'. Результат работы такой программы можно увидеть на рисунке 1.5:

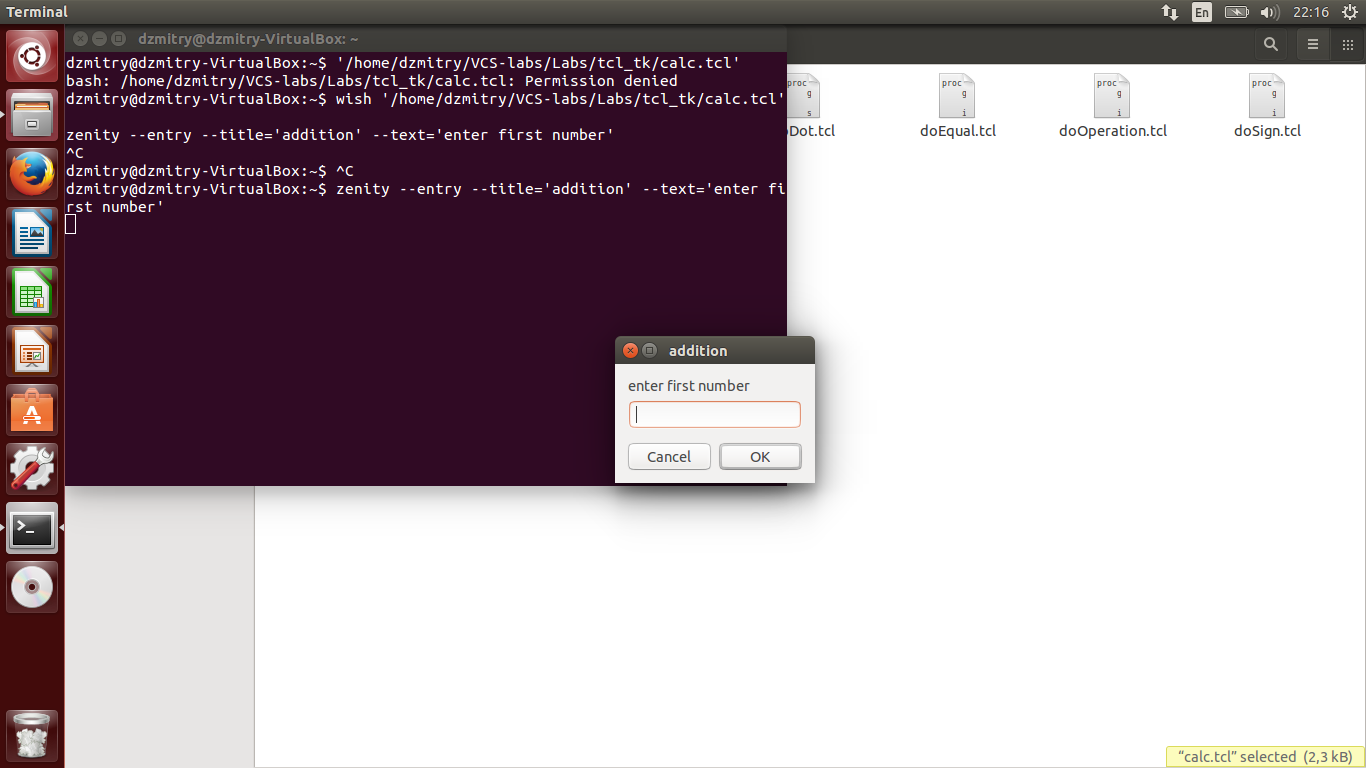


Рисунок 1.5 – Пример программы, созданной при помощи утилиты zenity

Таким образом, всего при помощи одной строки можно создать окно, которое выполнит какую-либо информационную функцию (или функцию ввода-вывода), и именно поэтому данная утилита очень хорошо подходит именно для первого знакомства с системами контроля версий.

В лабораторной работе по Subversion будет предложено создать простейшую программу-калькулятор, которая будет использовать простейшие математические операции, используя zenity. Данное задание было выбрано для того, чтобы можно было легко распараллелить работу каждого учащегося: каждому из них будет предложено создать собственный файл, реализовать арифметическую операцию согласно варианту, а потом положить всё получившееся на удалённый сервер.

**1.5 Концепция практикума по децентрализованным системам**

Кроме централизованных систем в предложенном лабораторном практикуме мы будем использовать децентрализованные системы контроля версий (ДСКВ или DVCS – Distributed Version Control System).

Они были сделаны для того, чтобы решить проблему с единой точкой отказа у централизованных систем контроля версий и реализовать некоторые функции, которые невозможно было бы сделать в централизованных системах. В качестве примеров можно привести следующие ДСКВ:

− Git;

− Mercurial;

− Darcs.

Как следует из названия, в таких системах клиенты не просто забирают последнюю (или выбранную, но всегда одну) версию проекта: на самом деле они полностью копируют его содержание и историю. Таким образом, при таком подходе у системы появляются два больших преимущества перед VCS. Первое заключается в том, что в любой момент времени разработчик может посмотреть состояние проекта в какой-то момент времени даже без наличия доступа к центральному серверу. Второй плюс в том, что каждая копия проекта это, по сути дела, полная резервная копия проекта. Таким образом, если сервер выйдет из строя, то любой разработчик сможет восстановить его состояние на любой другой машине.

Диаграмма взаимоотношений между клиентом и сервером в случае с ДСКВ представлена на рисунке 1.6.



Рисунок 1.6 – Децентрализованная система контроля версий