**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.

Данная проблема возникает при недостаточном планировании структуры (или архитектуры) будущего программного продукта, что чаще всего является следствием отсутствия достаточного финансирования проекта: программа нужна, сколько времени займёт разработка, каковы этапы, можно ли какие-то этапы исключить или сэкономить – следствием этого процесса является то, что этап проектирования сокращается;

− недостаток контроля. Без точной оценки процесса разработки срываются графики выполнения работ и превышаются установленные бюджеты. Сложно оценить объем выполненной и оставшейся работы.

Данная проблема возникает на этапе, когда проект, завершённый более, чем на половину, продолжает разрабатываться после дополнительного финансирования без оценки степени завершённости проекта;

− недостаток трассировки;

− недостаток мониторинга. Невозможность наблюдать ход развития проекта не позволяет контролировать ход разработки в реальном времени. С помощью инструментальных средств менеджеры проектов принимают решения на основе данных, поступающих в реальном времени.

Данная проблема возникает в условиях, когда стоимость обучения менеджмента владению инструментальными средствами, сравнима со стоимостью разработки самой программы;

− неконтролируемые изменения. У потребителей постоянно возникают новые идеи относительно разрабатываемого программного обеспечения. Влияние изменений может быть существенным для успеха проекта, поэтому важно оценивать предлагаемые изменения и реализовывать только одобренные, контролируя этот процесс с помощью программных средств.

Данная проблема возникает вследствие нежелания конечного потребителя использовать те или иные программные среды. Например, когда при создании клиент-серверной системы потребитель предъявляет требования не только к операционной системе на компьютерах-клиентах, но и на компьютере-сервере;

− недостаточная надежность. Самый сложный процесс – поиск и исправление ошибок в программах на ЭВМ. Поскольку число ошибок в программах заранее неизвестно, то заранее неизвестна и продолжительность отладки программ, и отсутствие гарантий отсутствия ошибок в программах.

Данная проблема возникает при неправильном выборе средств разработки. Например, при попытке создать программу, требующую средств высокого уровня, с помощью средств низкого уровня. Например, при попытке создать средства автоматизации с СУБД на ассемблере. В результате исходный код программы получается слишком сложным и плохо поддающимся структурированию;

− отсутствие гарантий качества и надежности программ из-за отсутствия гарантий отсутствия ошибок в программах вплоть до формальной сдачи программ заказчикам.

Качество программного обеспечения — это характеристика программного обеспечения как степени его соответствия требованиям. При этом требования могут трактоваться довольно широко, что порождает целый ряд независимых определений понятия. Чаще всего используется определение ISO 9001, согласно которому качество есть «степень соответствия присущих характеристик требованиям».

**1.2 Качество исходного кода**

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.

**1.3 Факторы качества**

В первую очередь надо выяснить по каким метрикам надо определять качество кода и для чего это нам вообще нужно. В программировании, в большинстве случаев, для определения метрики нам достаточно две важные для нас характеристики:

− соответствие правилам.

Под этот пункт подпадают ситуации, когда код компилируется и, в большинстве случаев, работает. Но зачем нужны правила написания кода, если код делает свое дело? 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. Этот список можно продолжить. Когда программист пишет код, он его легко читает, потому что он знает свой собственный стиль. Но стоит ему дать код, где используется другая нотация и скобки с новой строки, ему придется тратить дополнительное внимание на восприятие нового стиля. Особенно плохи в разработке ситуации, когда несколько совсем разных стилей используются в одном проекте или даже модуле.

− правила поддержки кода — правила, которые должны сигнализировать что код слишком сложный и его будет трудно сопровождать. К примеру, индекс сложности (подробнее о нем ниже) метода или класса слишком большой или слишком много строк кода в методе, наличие дубликатов в коде или “магических цифр”. Все они указывают нам на узкие места, которые сложно будет сопровождать. Но нельзя забывать, что именно мы можем решить какой индекс сложности для нас большой, а какой приемлемый.

− 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. Метрика здесь очевидная: соответствие правилам должно стремится к 100%, то есть чем меньше нарушений правил, тем лучше.

Использование систем контроля версий позволяет не только избавиться от лишнего кода, но и улучшить его качество, так как в любой момент времени можно узнать, кто последний работал над той или иной строкой.

− 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. Стоит мерить сложность метода, класса, файла. Значение этой метрики надо ограничить неким предельным числом. К примеру цикломатическая сложность метода не должна превышать 10, иначе нужно упростить или разбить его.

Использование SVN-возможностей, а именно комментирование версий, помогает снизить сложность кода.

**1.4 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, т. н. 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.

**1.5 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 [5]:

− Каскадная модель. 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 1.2;

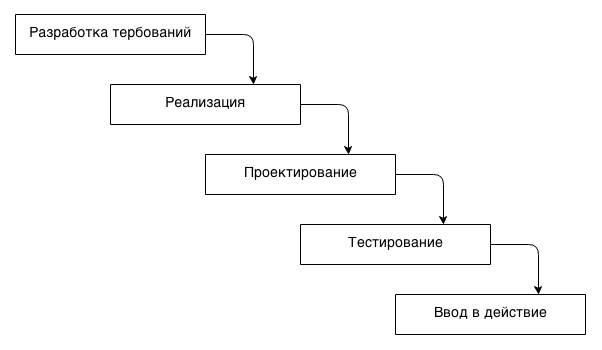


Рисунок 1.2 – Каскадная модель

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.

− Поэтапная модель с промежуточным контролем. Разработка ведется итеративно с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующие влияния результатов разработки друг на друга на различных этапах. Пример поэтапной модели с промежуточным контролем изображен на рисунке 1.3;

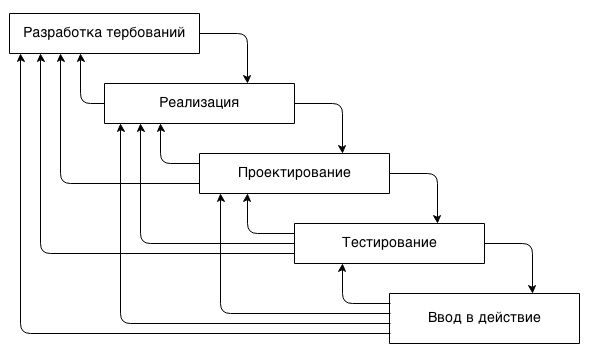


Рисунок 1.3 – Поэтапная модель с промежуточным контролем

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.

− Спиральная модель. 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 спирально модели изображен на рисунке 1.4.

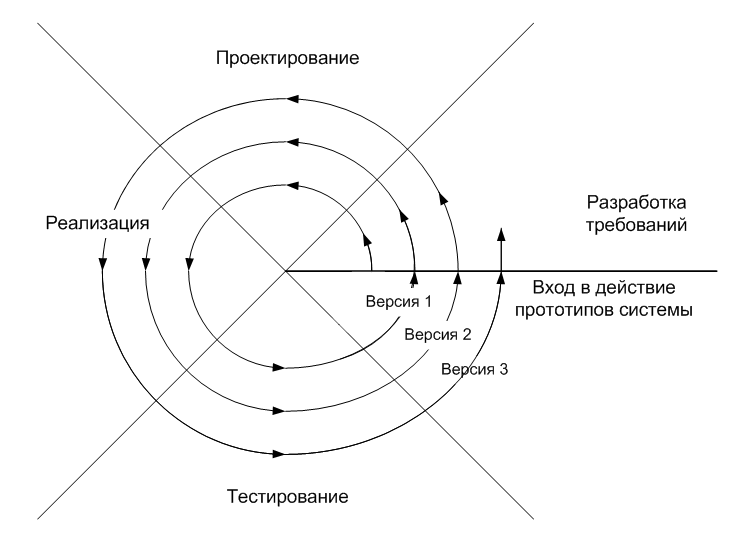


Рисунок 1.4 – Спиральная модель

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.