**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 нет «хорошо написанного» или «плохо написанного» кода. Но то, как написан код, может сильно влиять 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). 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 определить важную для нас характеристику:

− соответствие правилам;

− сложность кода;

− дубликаты;

− комментирование;

− покрытие тестами.

Нас интересуют первые два.

Соответствие правилам: под этот пункт подпадают ситуации, когда код компилируется и, в большинстве случаев, делает свое дело, при чем делает это правильно. Это интересная характеристика в большей степени от того, что в компании сначала должны существовать правила написания кода. Можно поступить проще и взять труд других (Java Code Conventions, GCC Coding Conventions, Zends Coding Standard), а можно поработать и дополнить их своими, наиболее подходящими для специфики вашей компании. Но зачем нам правила написания кода, если код делает свое дело? Чтобы ответить на вопрос выделим несколько типов правил:

− синтаксические правила — одни из наиболее бесполезных правил (но только первый взгляд), поскольку совсем никоим образом не виляют на исполнение программы. К ним можно отнести стиль именования переменных (camelCase, через подчеркивание), констант (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 венгерская нотация и скобки с новой строки, ему придется тратить дополнительное внимание на восприятие нового стиля. Особенно веселит ситуация, 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 “magic numders”. 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 в том смысле, что редко кто-то будет утверждать, что выражения очень нужны, даже когда они нигде не используются. Сюда можно отнести лишние импорты, переменные и методы, которые уже не используются, но по какой-то причине их оставили в наследство. Метрика здесь очевидная: соответствие правилам должно стремится к 100%, то есть чем меньше нарушений правил, тем лучше.

Цикломатическая сложность кода – характеристика, от которой напрямую зависит сложность поддержки кода. Здесь выделить метрику посложнее чем в предыдущей характеристике. Если по-простому, оно зависит от количества вложенных операторов ветвления и циклов. Кому интересно более подробное описания, можно почитать на [вики](http://en.wikipedia.org/wiki/Cyclomatic_complexity). 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 git-возможностей с комментированием версий помогает снизить сложность кода.

**1.4 Использование систем контроля версий программного обеспечения**

Преимущества и недостатки git по сравнению с централизованными системами управления версиями (такими как, например, [Subversion](https://ru.wikipedia.org/wiki/Subversion" \o "Subversion)) типичны для любой распределённой системы. Если же сравнивать git с «родственными» ей распределёнными системами, можно отметить, что git изначально идеологически ориентирован на работу с изменениями, а не с файлами, «единицей обработки» для него является набор изменений, или патч. Эта особенность прослеживается как в структуре самой системы (в частности — в структуре репозитория), так и в принципах построения команд; она отражается на производительности системы в различных вариантах её использования и на достоинствах и недостатках git по сравнению с другими [DVCS](https://ru.wikipedia.org/wiki/DVCS).

Часто называемые преимущества git перед другими DVCS:

− Высокая производительность.

− Развитые средства интеграции с другими [VCS](https://ru.wikipedia.org/wiki/VCS), в частности, с [CVS](https://ru.wikipedia.org/wiki/CVS), [SVN](https://ru.wikipedia.org/wiki/SVN) и [Mercurial](https://ru.wikipedia.org/wiki/Mercurial" \o "Mercurial). Помимо разнонаправленных конвертеров репозиториев, имеющиеся в комплекте программные средства позволяют разработчикам использовать git при размещении центрального репозитория в SVN или CVS, кроме того, git может имитировать cvs-сервер, обеспечивая работу через клиентские приложения и поддержку в средах разработки, специально не поддерживающих git.

− Продуманная система команд, позволяющая удобно встраивать git в скрипты.

− Качественный веб-интерфейс «из коробки».

− Репозитории git могут распространяться и обновляться общесистемными файловыми утилитами архивации и обновления, такими как [rsync](https://ru.wikipedia.org/wiki/Rsync" \o "Rsync), благодаря тому, что фиксации изменений и синхронизации не меняют существующие файлы с данными, а только добавляют новые (за исключением некоторых служебных файлов, которые могут быть автоматически обновлены с помощью имеющихся в составе системы утилит). Для раздачи репозитория по сети достаточно любого веб-сервера.

В числе недостатков git обычно называют:

− Отсутствие сквозной нумерации коммитов монотонно непрерывно возрастающими целыми числами. Во многих проектах используется автоматические получение номера этой версии (например, командой svn version), построение .H файла на основе этого числа, и далее его использование при создании штампа версии исполняемого файла, некоторых вшитых в него строк и так далее.

− Отсутствие переносимой на другие операционные системы поддержки путей в кодировке Unicode в Microsoft Windows (для версий msysgit до 1.8.1). Если путь содержит символы, отличные от [ANSI](https://ru.wikipedia.org/wiki/ANSI), то их поддержка из командной строки требует специфических настроек, которые не гарантируют правильного отображения файловых имён при пользовании тем же репозиторием из других ОС. Одним из способов решения проблемы для git 1.7 является использование специально пропатченного консольного клиента. Другой вариант — использование графических утилит, работающих напрямую через [API](https://ru.wikipedia.org/wiki/API), таких как [TortoiseGit](https://ru.wikipedia.org/wiki/TortoiseGit" \o "TortoiseGit).

− Некоторое неудобство для пользователей, переходящих с других VCS. Команды git, ориентированные на наборы изменений, а не на файлы, могут вызвать недоумение у пользователей, привыкших к файл-ориентированным VCS, таким как SVN. Например, команда «add», которая в большинстве систем управления версиями производит добавление файла к проекту, в git подготавливает к фиксации сделанные в файлах изменения. При этом сохраняется не патч, описывающий изменения, а новая версия целевого файла.

− Использование для идентификации ревизий хэшей [SHA1](https://ru.wikipedia.org/wiki/SHA1), что приводит к необходимости оперировать длинными строками вместо коротких номеров версий, как во многих других системах (хотя в командах допускается использование неполных хэш-строк).

− Бо́льшие накладные расходы при работе с проектами, в которых делаются многочисленные несвязанные между собой изменения файлов. При работе в таком режиме размеры наборов изменений становятся достаточно велики и происходит быстрый рост объёма репозиториев.

− Бо́льшие затраты времени, по сравнению с файл-ориентированными системами, на формирование истории конкретного файла, истории правок конкретного пользователя, поиска изменений, относящихся к заданному месту определённого файла.

− Отсутствие отдельной команды переименования/перемещения файла, которая отображалась бы в истории как соответствующее единое действие. Существующий скрипт git mv фактически выполняет переименование, копирование файла и удаление его на старом месте, что требует специального анализа для определения, что в действительности файл был просто перенесён (этот анализ выполняется автоматически командами просмотра истории). Однако, учитывая тот факт, что наличие специальной команды для переименования/перемещения файлов технически не вынуждает пользователя использовать именно её (и, как следствие, в этом случае возможны разрывы в истории), поведение git может считаться преимуществом.

− Система работает только с файлами и их содержимым, и не отслеживает пустые каталоги.

В ряде публикаций, относящихся преимущественно к 2005—2008 годам можно встретить также нарекания в отношении документации git, отсутствия удобной windows-версии и удобных графических клиентов. В настоящее время эта критика неактуальна: существует версия git на основе [MinGW](https://ru.wikipedia.org/wiki/MinGW" \o "MinGW) («родная» сборка под Windows), и несколько высококачественных графических клиентов для различных операционных систем, в частности, под Windows имеется клиент [TortoiseGit](https://ru.wikipedia.org/wiki/TortoiseGit" \o "TortoiseGit), идеологически очень близкий к широко распространённому [TortoiseSVN](https://ru.wikipedia.org/wiki/TortoiseSVN" \o "TortoiseSVN) — клиенту SVN, встраиваемому в оболочку Windows.

**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;

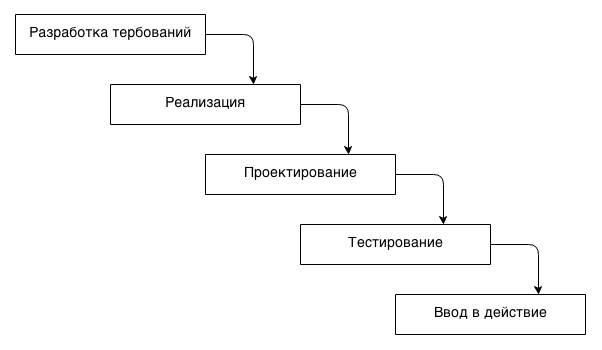


Рисунок 1.2 – Каскадная модель

− 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. Вследствие этого время жизни каждого из этапов растягивается практически на все время разработки. Пример поэтапной модели с промежуточным контролем изображен на рисунке 1.3;

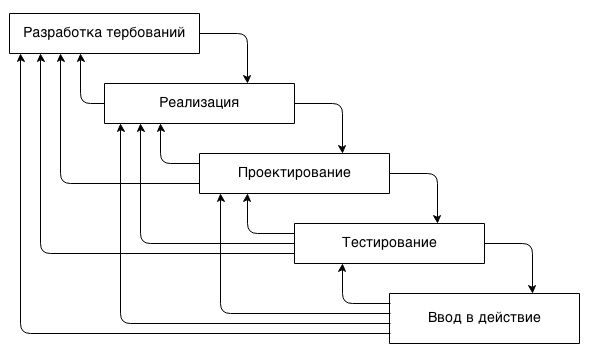


Рисунок 1.3 – Поэтапная модель с промежуточным контролем

− 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.

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.

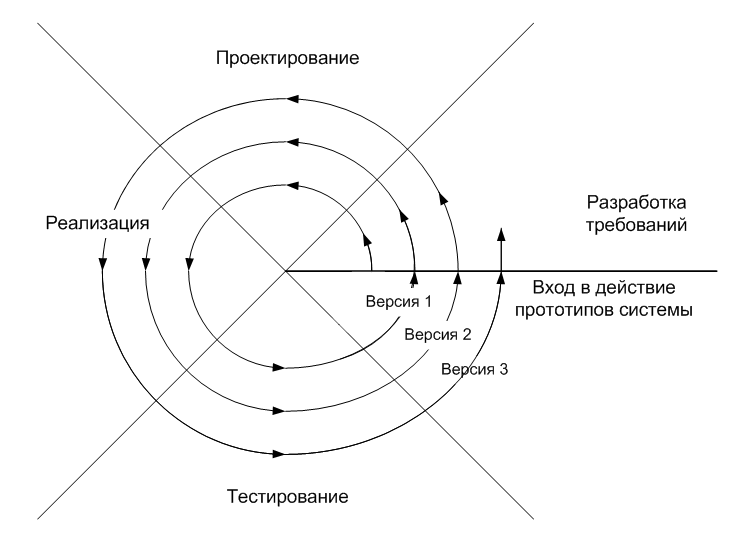


Рисунок 1.4 – Спиральная модель