Технология программирования

Жизненный цикл и современные методы и процессы разработки ПО

Вопрос 1. Понятие жизненного цикла ПО

Информационные системы должны соответствовать потребностям бизнеса, легко адаптироваться и быть экономичными. Если систему спроектировали плохо, то поддерживать и обновлять ее будет долго и дорого. Поэтому один из главных принципов подхода к созданию информационных систем — это идея жизненного цикла программного обеспечения.

Жизненный цикл программного обеспечения (ПО) — это непрерывный процесс, который начинается с решения, что надо разработать ПО, и завершается полной деоперационализацией уже созданного.

Для создания программного обеспечения применяют специализированный алгоритм, известный как SDLC (Software Development Life Cycle), или жизненный цикл ПО. Это основополагающая структура, которая упрощает разработку и поддержку крупных IT-проектов за счет последовательности операций.

Программы «большие» и «маленькие»

Каждый, кто хотя бы раз создавал программу, может представить процесс разработки «небольшого» приложения, предназначенного, чтобы решить конкретную и несложную задачу. Такие программы в основном создаются для одного пользователя или ограниченной группы людей.

«**Небольшие» программы** обычно имеют объем около 150 строк кода и решают четко сформулированную задачу в известных рамках. Неважно, с какой скоростью выполняются эти программы, а потенциальный негативный эффект от их неправильной работы минимален. Исключение составляют ситуации, когда сбой программы влияет на систему, которая выполняет более важные задачи. К таким приложениям не нужно добавлять новые функции, обновлять их и исправлять найденные ошибки как правило не надо.

Сложные программы, или программные системы, комплексы или продукты, отличаются от менее масштабных по объему. Кроме того, есть факторы,

которые влияют на то, хотят ли пользователи приобретать, сопровождать и обучать их.

Характеристики комплексной программы

1. Она решает одну или несколько взаимосвязанных задач, у которых нет четкой формулировки на начальном этапе, но они критически важны для отдельных пользователей или организаций и приносят им значительные выгоды, когда эти программы используют.
2. Ей удобно пользоваться. Это означает, что есть исчерпывающая и понятная документация для пользователей, а также специализированные материалы для администраторов и обучения.
3. Низкая производительность при обработке реальных данных может вести к значительным потерям для пользователей.
4. Неправильная работа программного обеспечения может причинить серьезный ущерб пользователям и третьим лицам, даже если сбои происходят относительно редко.
5. Для достижения своих целей программа должна уметь взаимодействовать с другими ПО и программно-аппаратными системами, а также функционировать на разных платформах.
6. Пользователи получают дополнительные преимущества, когда продолжают ее развивать, а значит появляются новые функции, исправляются ошибки.
7. Нужна проектная документация, которая позволяет развивать систему даже тем разработчикам, которые не участвовали в ее создании, без значительных затрат на реинжиниринг.
8. Процесс разработки включает большое количество специалистов, обычно более пяти человек.

Создать «большую» программу с первой попытки практически невозможно, это требует значительных усилий, в разработке участвуют несколько человек.

Принципы работы со сложными системами

Когда проектируют сложные системы, помимо методических рекомендаций важно применять прагматические принципы, которые делают процесс более эффективным. Эти принципы помогают разрабатывать качественные технические решения в широком диапазоне приложений.

Они позволяют оптимально распределять задачи между участниками проектов, минимизируя затраты на коммуникацию, и акцентировать внимание на наиболее критичных особенностях работы каждого из них.

Принципы работы со сложными системами

1. Абстракция.
2. Уточнение.
3. Модульная разработка.
4. Повторное использование компонентов.

Абстракция представляет собой мощный инструмент для анализа сложных систем. Индивидуальные возможности восприятия ограничены, и человек не может одновременно обрабатывать все элементы и характеристики высокосложных систем. Для плавного перехода от абстрактного к конкретному применяется метод последовательного **уточнения**, который помогает выявить и проработать детали. Абстракцию и уточнение используют потому, что это помогает создать решения, которые эффективно функционируют и соответствуют нужным характеристикам конечной системы.

Примеры абстракции и уточнения

Пример 1. Систему хранения идентификаторов пользователей интернет-магазина можно представить, как множество целых чисел, если забыть о том, что эти числа — идентификаторы пользователей, и о том, что все это как-то связано с интернет-магазином. Затем описанную модель системы хранения идентификаторов пользователей интернет-магазина можно уточнить, определить конкретную реализацию множества чисел, например, на основе сбалансированных красно-черных деревьев.

Пример 2. В этом примере речь идет о том, как можно передать данные по сети, если абстрагироваться от множества других проблем связи и сконцентрироваться только на одной задаче — как передавать данные надежно и в правильном порядке.

Предположим, что мы уже можем передавать данные между двумя компьютерами, но с некоторыми потерями и нарушением порядка доставки. Чтобы обеспечить надежную передачу данных в нужной последовательности, нам нужен специальный набор правил — протокол транспортного уровня, например, TCP. Этот протокол помогает решить конкретную задачу — доставить данные без потерь и в правильном порядке.

Однако, для того чтобы транспортный протокол вообще мог работать, нужно, чтобы на более низких уровнях сети (сетевом, канальном и физическом) существовали свои протоколы. Эти протоколы решают задачи организации связи между компьютерами, формата данных и передачи сигналов, то есть обеспечивают базовые условия для того, чтобы данные вообще могли передаваться по сети.

Проще говоря, сначала мы разбираемся с основным вопросом — как сделать передачу данных надежной, а затем углубляемся в более технические детали, которые позволяют организовать саму возможность передачи данных между компьютерами.

Модульность — это принцип организации крупных систем, когда их разбивают на набор подсистем, модулей или компонентов.

Этот подход предполагает, что сложная система формируется как совокупность более простых систем — модулей, которые взаимодействуют друг с другом через интерфейсы. Каждая задача, которую решает вся система, разбивается на более простые подзадачи, которые решают отдельные модули. Мы комбинируем их решения и получаем решение исходной задачи.

После того, как задача разбита на подзадачи, а для каждой подзадачи определены модули, то есть отдельные части системы, можно анализировать каждую часть отдельно. После этого нужно подумать, как соединить эти модули в единую систему, чтобы она решала изначальные задачи.

Если для взаимодействия между модулями заранее прописать четкие правила (интерфейсы), то это значительно упрощает процесс сборки всей системы. Это позволяет разработчикам сосредоточиться на том, что именно делают интерфейсы, и не вникать в сложные детали работы каждого модуля. Такой подход предотвращает излишнюю сложность системы.

Приведем пример. Он объясняет, как устроена система передачи данных по сети с помощью протоколов. Основная идея в том, что всю систему передачи данных удобно разбить на несколько уровней, где каждый уровень решает свои конкретные задачи. Например, один уровень может отвечать за то, как данные передаются по сети, а другой — за то, чтобы они доходили до адресата в правильном порядке.

Каждый уровень взаимодействует как с верхним уровнем, которому он предоставляет свои услуги, например, передает данные дальше, так и с нижним уровнем, от которого он получает нужные для работы ресурсы. Проще говоря, каждый уровень системы может обращаться к нижним уровням за помощью, чтобы выполнить свои задачи.

Преимущество такой системы в том, что, если нам нужно заменить или улучшить какой-то протокол на одном уровне, это не требует изменений на других уровнях. Это делает систему гибкой и легко модернизируемой. Однако правильное разделение системы на такие уровни — это сложная задача, которая требует дополнительных правил и принципов.

Таким образом, речь в примере идет о том, как с помощью уровневого подхода к протоколам передачи данных можно упростить их разработку и поддержку.

Вопрос 2. Стандарты жизненного цикла

Чтобы понять возможную структуру жизненного цикла ПО, обратимся сначала к соответствующим стандартам, которые описывают технологические процессы.

Международными организациями разработан набор стандартов, которые регламентируют разные стороны жизненного цикла и вовлеченных в него процессов.

Международные организации:

- IEEE — читается «ай-трипл-и», Institute of Electrical and Electronic Engineers, Институт инженеров по электротехнике и электронике;
- ISO — International Standards Organization, Международная организация по стандартизации;
- EIA — Electronic Industry Association, Ассоциация электронной промышленности;
- IEC — International Electrotechnical Commission, Международная комиссия по электротехнике;
- ANSI — American National Standards Institute, Американский национальный институт стандартов;
- SEI — Software Engineering Institute, Институт программной инженерии;
- ECMA — European Computer Manufactures Association, Европейская ассоциация производителей компьютерного оборудования.

Рассмотрим список и общее содержание этих стандартов.

Группа стандартов ISO

Стандарт ISO/IEC 12207 для информационных технологий. Процессы жизненного цикла программного обеспечения (аналог ГОСТ Р ИСО/МЭК 12207-2010 в России). Устанавливает общую структуру жизненного цикла ПО, которая представлена в виде трехступенчатой модели.

Эта модель включает процессы, виды деятельности и задачи. Стандарт описывает элементы, фокусируется на их целях и ожидаемых результатах. Подразумевает возможные взаимосвязи, хотя точную структуру этих связей не определили. Также не указали способы организации элементов в проекте и метрики для оценки прогресса и эффективности. Ключевыми компонентами

считаются процессы жизненного цикла программного обеспечения, которых насчитывается 18, и они разделены на 4 категории.

Таблица 1. Стандарт ISO/IEC 12207

Основные процессы	• Приобретение ПО; • Передача ПО (в использование); • Разработка ПО; • Эксплуатация ПО; • Поддержка ПО
Поддерживающие процессы	• Документирование; • Управление конфигурациями; • Обеспечение качества; • Верификация; • Валидация; • Совместные экспертизы; • Аудит; • Разрешение проблем
Организационные процессы	• Управление проектом; • Управление инфраструктурой; Усовершенствование процессов; • Управление персоналом
Адаптация	• Адаптация описываемых стандартом процессов под нужды конкретного проекта

ISO/IEC 15288 Standard for Systems Engineering — System Life Cycle Processes. Процессы жизненного цикла систем (ГОСТ Р ИСО/МЭК 15288-2005). Отличается от предыдущего тем, что должен рассматривать программно-аппаратные системы в целом. В настоящий момент этот стандарт все еще приводят в соответствие с предыдущим. ISO/IEC 15288 предлагает рассматривать жизненный цикл системы по похожей системе, в виде набора процессов. Каждый процесс описывается набором его результатов (outcomes), которые достигаются при помощи разных видов деятельности. Всего выделено 26 процессов, которые объединяются в 5 групп.

Таблица 2. ISO/IEC 15288

Процессы выработки соглашений	• Приобретение системы; • Поставка системы
Процессы уровня организации	• Управление окружением; • Управление инвестициями; • Управление процессами; • Управление ресурсами;

	• Управление качеством
Процессы уровня проекта	• Планирование; • Оценивание; • Мониторинг; • Управление рисками; • Управление конфигурацией; • Управление информацией; • Выработка решений
Технические процессы	• Определение требований; • Анализ требований; • Проектирование архитектуры; • Реализация; • Интеграция; • Верификация; • Валидация; • Передача в использование; • Эксплуатация; • Поддержка; • Изъятие из эксплуатации
Специальные процессы	• Адаптация описываемых стандартом процессов под нужды конкретного проекта

Группа стандартов IEEE

IEEE 1074-1997 — стандарт IEEE для разработки процессов жизненного цикла программного обеспечения. Он создан, чтобы создавать индивидуализированные процессы разработки для конкретного проекта. Документ описывает требования, которым должен соответствовать любой процесс, а также общую структуру процесса разработки. В этой структуре определяются ключевые виды деятельности, задействованные в процессах, а также перечень входящих и исходящих документов, которые нужны, чтобы выполнить эти виды деятельности. В рамках стандарта рассматриваются пять подпроцессов, 17 групп деятельностей и 65 видов деятельности.

К примеру, подпроцесс разработки включает группы деятельностей по сбору требований, проектированию и реализации. Группа деятельностей по проектированию охватывает:

- архитектурное проектирование;
- проектирование баз данных;
- проектирование интерфейсов;

- детальное проектирование компонентов.

IEEE/EIA 12207-1997 — IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Software Life Cycle Processes. Промышленное использование стандарта ISO/IEC 12207 на процессы жизненного цикла ПО. Аналог ISO/IEC 12207, сменил стандарты J-Std-016-1995 EIA/IEEE Interim Standard for Information Technology — Software Life Cycle Processes — Software Development Acquirer-Supplier Agreement. Промежуточный стандарт на процессы жизненного цикла ПО и соглашения между поставщиком и заказчикам ПО.

Группа стандартов CMM, разработанных SEI

Модель зрелости возможностей CMM (Capability Maturity Model) предоставляет систематизированный подход к оценке способности организации выполнять задачи разной сложности. Она делит процессы на три основные категории: уровни зрелости организации, ключевые области процесса и ключевые практики. Обычно, когда говорят о модели CMM, имеют в виду именно модель уровней зрелости.

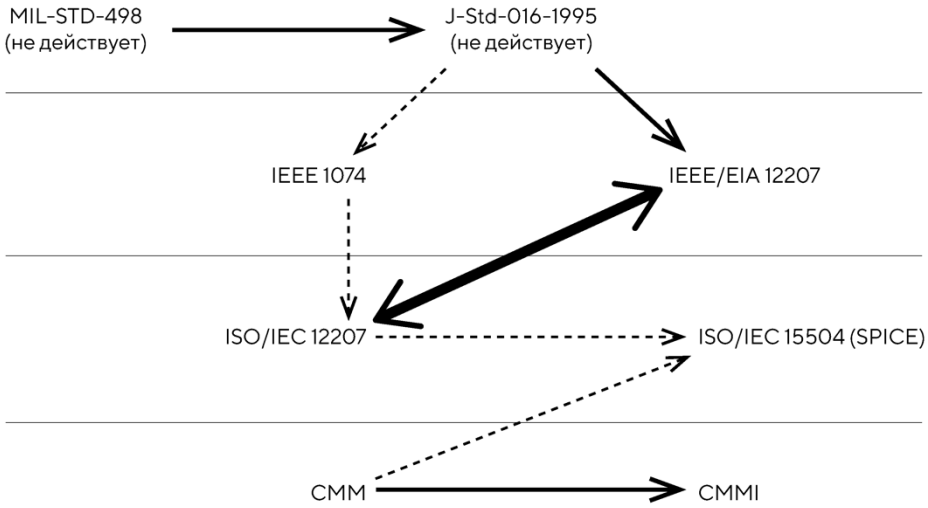
Однако на сегодняшний день CMM считается устаревшей и постепенно заменяется моделью CMMI. По шкале CMM, уровни зрелости определяются тем, что применяются четко структурированные техники и процедуры, которые относятся к разным ключевым областям процесса.

Каждая из этих областей представляет собой группу взаимосвязанных действий, цель которых — достичь результатов, критически важных для общей оценки эффективности технологических процессов. Выделяется в общей сложности 18 таких областей, и их количество увеличивается по мере продвижения организации на более высокие уровни зрелости.

Интегрированная модель зрелости возможностей CMMI (Capability Maturity Model Integration). Эта модель представляет собой результат того, как интегрировали модели CMM для продуктов и процессов, а также чтобы разрабатывали ПО и программно-аппаратные системы.

В целом перечисленные стандарты связаны так, как изображается на рисунке. Сплошные стрелки указывают направления исторического развития, жирная стрелка обозначает идентичность, пунктирные стрелки показывают влияние одних стандартов на другие.

Схема 1. Связь стандартов жизненного циклов ПО



Вопрос 3. Модели жизненного цикла

Существует несколько моделей жизненного цикла программного обеспечения. Наиболее распространенные:

- каскадная;
- инкрементная;
- спиральная.

Используются реже из-за своей специфической природы и ограниченной универсальности:

- v-образная;
- итеративная.

Далее рассмотрим ключевые характеристики некоторых из этих моделей.

Каскадная модель — это последовательный цикл этапов, где каждый уровень сменяет предыдущий в строгой последовательности, которую нельзя изменить.

Эта модель подходит, чтобы разрабатывать относительно простые программные продукты, для которых можно заранее определить четкий список требований.

Инкрементная модель включает линейную последовательность действий, обеспечивает на каждом этапе обратную связь и контроль достигнутых результатов.

В процессе реализации проекта создается несколько версий продукта, которые называются **инкрементами**. Применение этой модели позволяет одновременно финансировать крупные проекты поэтапно, находить дополнительные ресурсы и внедрять продукт частями. В таком случае пользователи получают не окончательную версию с множеством скрытых недостатков, а тестовые версии, которые можно постепенно улучшать.

Спиральная модель — это инновационный подход, который объединяет проектирование и поэтапное прототипирование программного обеспечения, позволяет проверять жизнеспособность сложных и нестандартных технических решений.

Главная цель этой модели заключается в снижении рисков, которые могут повлиять на ход жизненного цикла проекта. Каждый «виток спирали» ассоциируется с новой рабочей версией, что дает возможность объективно оценивать реальность выполнения задач и качество выполнения проекта в целом. Эта схема помогает выявить серьезные ошибки и функциональные недостатки еще на ранней стадии разработки.

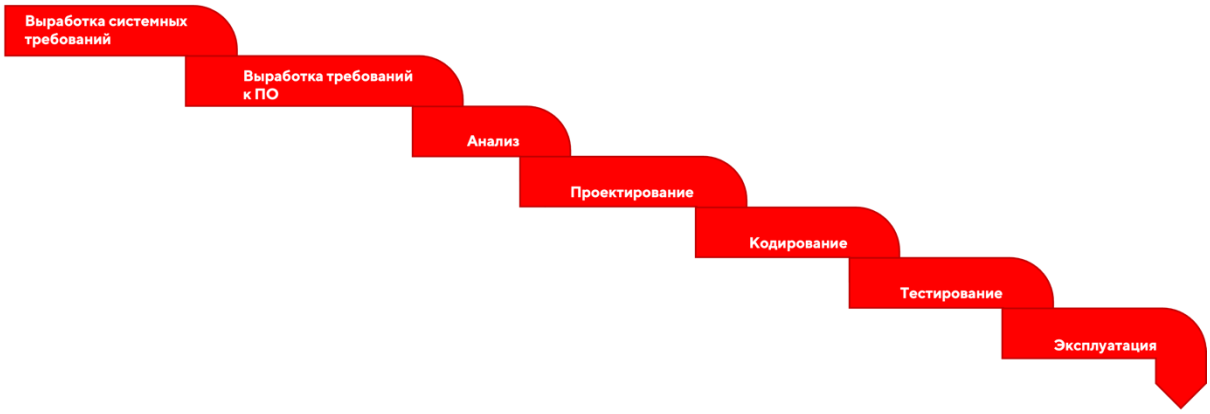
V-модель по сути представляет усовершенствованный вариант каскадной модели. Основное отличие заключается в том, что тестирование выполняется на каждом этапе разработки. Это помогает минимизировать количество ошибок в архитектуре программного обеспечения. Главный недостаток этой модели аналогичен недостаткам классической каскадной модели — отсутствует возможность допустить ошибку. Если разработчики допустят недочет на одном из этапов, исправить его будет трудно и дорого. Применение V-модели целесообразно при создании надежных и точных продуктов.

Итеративная модель. Преимущество этой модели в том, что она позволяет «ориентироваться на местности». Заранее определять закрытый список требований и составлять объемное техническое задание не нужно. Выявить актуальность и полезность продукта, а также возможные ошибки можно на этапе черновика.

Стандарты описывают, каким образом должен проходить процесс разработки программного обеспечения. Однако они часто используют слишком универсальные модели жизненного цикла, поэтому их трудно применять в рамках конкретных проектов. Наиболее известной и продолжительное время действующей считается **каскадная** (или водопадная) модель жизненного цикла.

На иллюстрации представлена «Классическая» каскадная модель, которая подразумевает строгое последовательное движение по этапам: все нужное для выполнения текущей стадии должно быть заранее подготовлено на предыдущих этапах. Тем не менее, эта модель не обязывает строго придерживаться установленного порядка выполнения работ.

Схема 2. Итеративная модель



Среди разработчиков и исследователей, которые создают сложное программное обеспечение, с самого начала существования этой индустрии высокую популярность приобрели модели эволюционных и итеративных процессов. Эти модели отличаются большей гибкостью и адаптивностью к условиям, которые быстро меняются. Итеративные или инкрементальные подходы (существует несколько вариантов таких моделей) предполагают разделение системы на отдельные компоненты, которые создаются через последовательные циклы выполнения всех нужных работ или их частей.

В ходе первой итерации разрабатывается компонент, который не зависит от других. В этом процессе проходят практически все этапы работ, после чего результаты анализируются. На следующей итерации либо вносятся изменения в уже созданный компонент, либо разрабатывается новый, который может зависеть от первого, либо происходит параллельная доработка первой части и добавляются новые функции. Таким образом, каждая итерация предоставляет возможность оценить промежуточные результаты и реакцию заинтересованных сторон, в том числе пользователей, что позволяет вносить коррективы на следующих циклах разработки.

Каждая итерация может охватывать полный спектр действий — от анализа требований до ввода в эксплуатацию новой части программного обеспечения.

Схема 3. Перечень итераций

Каскадная модель с возможностью возвращать на предшествующий шаг при необходимости пересмотреть его результаты, становится **итеративной**.

Спиральная модель жизненного цикла ПО развилась из идеи итераций. Ее ключевая особенность — это структура действий на каждом этапе, которая повторяется.

Эти действия включают планирование, определение задач и ограничений, поиск решений, оценку рисков, выполнение работ и анализ результатов.

Модель называется спиральной, потому что ход работ изображается в «полярных координатах»: угол показывает, какой этап выполняется, а расстояние от начала координат отражает, сколько ресурсов затрачено.

Схема 4. Спиральная модель