Программная архитектура

Назначение архитектуры программного обеспечения

Под архитектурой программного обеспечения понимают набор внутренних структур ПО, которые состоят из компонентов, их связей и возможных взаимодействий между компонентами, также доступных извне свойств этих компонентов. Под компонентом понимают достаточно произвольный структурный элемент программного обеспечения, который можно выделить, определив интерфейс взаимодействия между этим компонентом и всем, что его окружает.

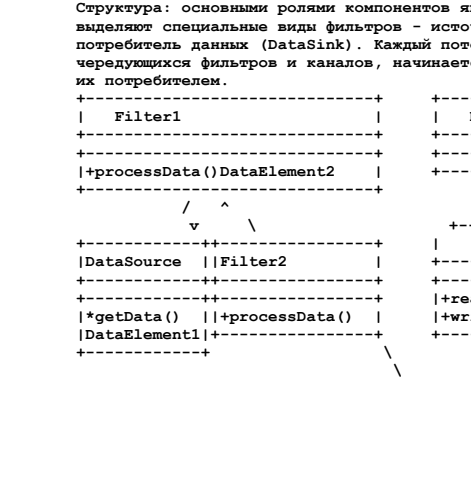
Архитектурный стиль может определяться архитектурным контуром, промежуточным программным обеспечением, рекомендованным набором шаблонов или методом архитектурного описания системы. Архитектурный стиль определяет основные правила выделения компонентов и организации взаимодействия между ними в рамках системы или подсистемы в целом.

Выбор архитектуры задаёт способ реализации требований на высоком уровне абстракции. Различные архитектурные стили подходят для решения различных задач в плане обеспечения нефункциональных требований, а именно: различных уровней производительности, удобства использования, переносимости и удобства сопровождения. Значительно меньшим является влияние архитектуры на функциональность, при этом заданную функциональность можно реализовать с использованием совершенно разных архитектур.

1) Pipes and Filters: предназначен для организации обработки потоков данных в том случае, когда процесс обработки распадается на несколько шагов. Эти шаги могут выполняться отдельными обработчиками, возможно, реализованными различными разработчиками или даже организациями.

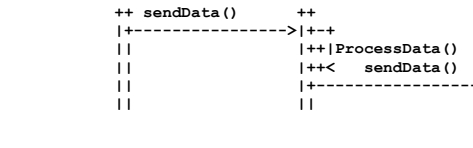
Решение: каждая отдельная задача по обработке данных разбивается на несколько мелких шагов, при этом выходные данные одного шага являются входными для других. Каждый шаг реализуется специальным компонентом, а именно фильтром. Фильтр потребляет и выдаёт данные инкрементально (небольшими порциями), передача данных между фильтрами осуществляется по каналам (pipes).

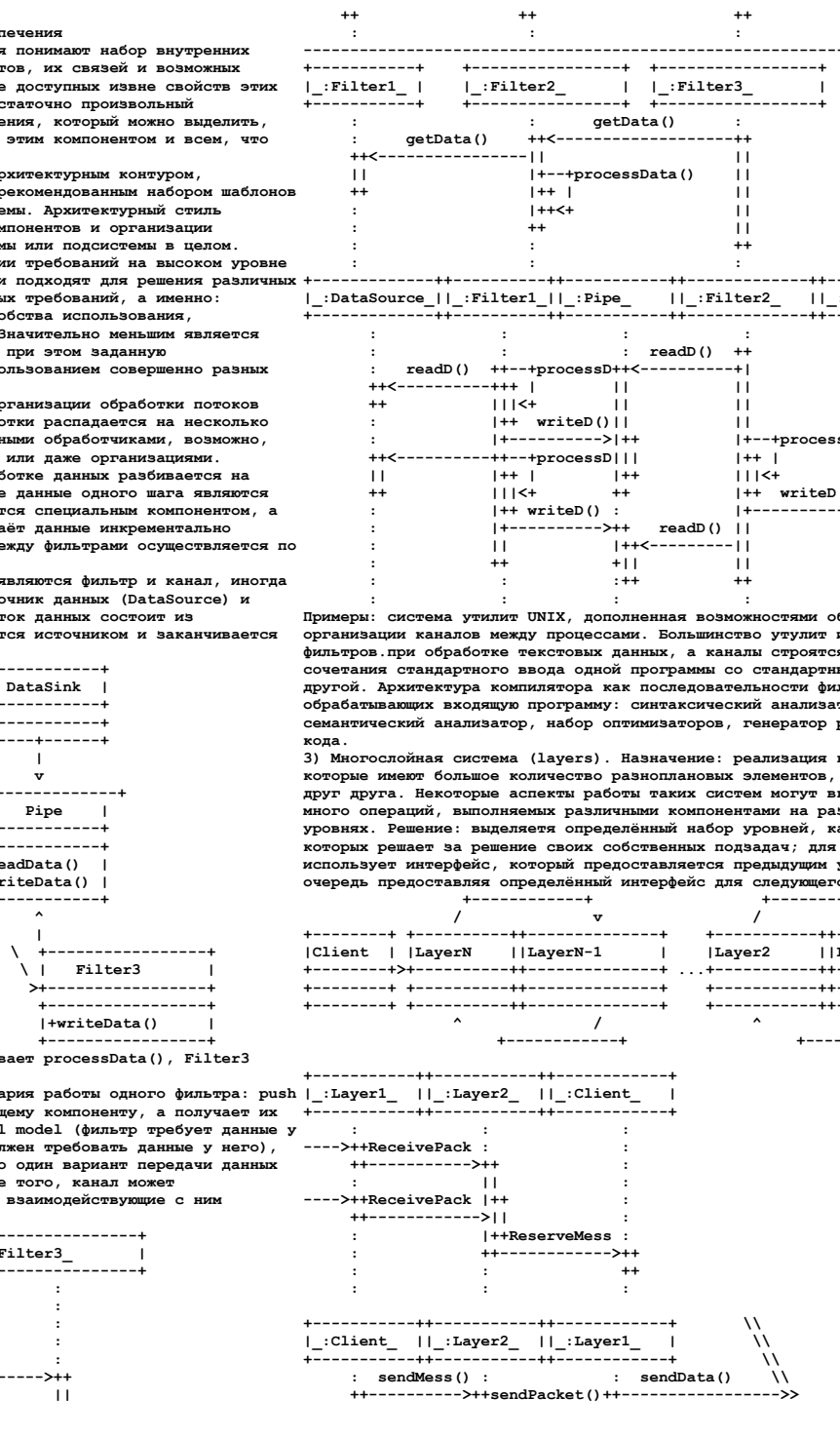
Структура: основными ролями компонентов являются фильтр и канал, иногда выделяют специальные виды фильтров - источник данных (DataSource) и потребитель данных (DataSink). Каждый поток данных состоит из чередующихся фильтров и каналов, начинается источником и заканчивается их потребителем.



Filter1 вызывает getData(), Filter2 вызывает processData(), Filter3 вызывает processData().

2) Динамика: возможны три различных сценария работы одного фильтра: push model (фильтр сам передаёт данные следующему компоненту, а получает их как результат передачи предыдущего), pull model (фильтр требует данные у предыдущего компонента, следующий сам должен требовать данные у него), push/pull model. Часто реализуется только один вариант передачи данных для всех фильтров в рамках системы, кроме того, канал может буферизировать данные и синхронизировать взаимодействующие с ним фильтры.

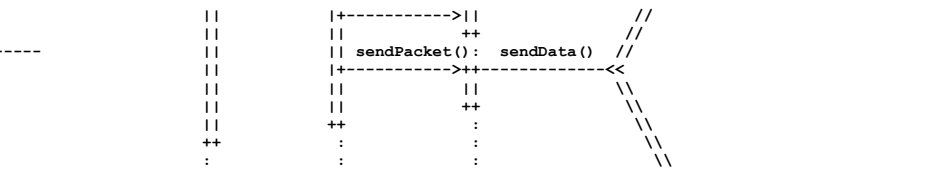




Примеры: система утилит UNIX, дополненная возможностями оболочки по организации каналов между процессами. Большинство утулит играют роль фильтров.при обработке текстовых данных, а каналы строятся с помощью сочетания стандартного ввода одной программы со стандартным выводом другой. Архитектура компилятора как последовательности фильтров, обрабатывающих входящую программу: синтаксический анализатор, семантический анализатор, набор оптимизаторов, генератор результирующего кода.

3) Многослойная система (layers). Назначение: реализация крупных систем, которые имеют большое количество разноплановых элементов, использующих друг друга. Некоторые аспекты работы таких систем могут включать в себя много операций, выполняемых различными компонентами на различных уровнях. Решение: выделяетя определённый набор уровней, каждый из которых решает за решение своих собственных подзадач; для этого он использует интерфейс, который предоставляется предыдущим уровнем, в свою очередь предоставляя определённый интерфейс для следующего уровня.

Часто в виде многих уровней реализуются коммуникационные системы. Две такие системы могут взаимодействовать через самый нижний уровень, при этом пара симметричных сценариев по подъёму-спуску обращений выполняется в рамках одного общего сценария на разных машинах. Обращение клиента к верхнему уровню инициирует цепочку обращений от верхнего уровня до самого нижнего. События на нижнем уровне (получение сообщения по сети. нажатие кнопки мыши) инициирует цепочку обращений снизу вверх, вплоть до события на верхнем уровне, которое является видимым клиентом. При этом бывают ситуации, когда обращение клиента к верхнему уровню не доходит до самого низа (один из уровней кэширует ответы на запросы и соответственно выдаёт результат без необходимости обращения к нижним уровням.



Примеры: стандартная система связи открытых систем (Open System Interconnection), архитектура современных информационных систем или систем автоматизации бизнеса; она содержит четыре уровня, а именно: интерфейс взаимодействия с внешней средой, бизнес-логика, предметная область и уровень управления ресурсами.

MVC (Модель-Вид-Контроллер)

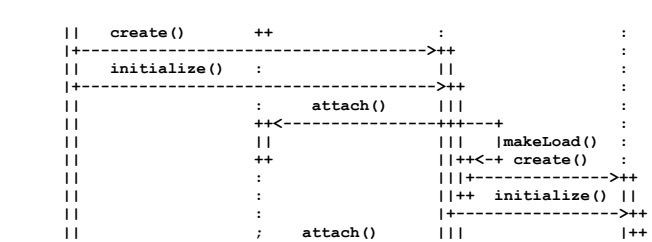
Назначение: когда различные пользователи имеют различные наборы требований, методы же обработки данных изменяются в меньшей степени. В таких случаях визуальное представление управляющих элементов может меняться вместе с интерфейсом, а сами выполняемые действия зависят от бизнес-логики. и предметной области и поэтому более устойчивы. Наиболее подлежит изменениям модель данных, поэтому для увеличения гибкости и удобства изменений в таких приложениях необходимо соответствующим образом выделить компоненты.

Решение: первый набор: данные, модель данных или просто модель (model): соответствует структуре данных предметной области, в которой работает приложение; обязанности этих компонентов представляют в системе данные и базовые операции над ними. Второй набор: представление (view): соответствует различным способам представления данных в интерфейсе пользователя, при этом для одних данных может быть несколько представлений; каждому компоненту представления в соответствие ставится компонент из третьего набора: обработчик (controller) - компонент, который осуществляет обработку действий пользователя. Такой компонент получает команды и превращает их в действия над данными. В результате каждого действия нужно обновить все представления всех данных, которые подверглись изменениям.

Структура:

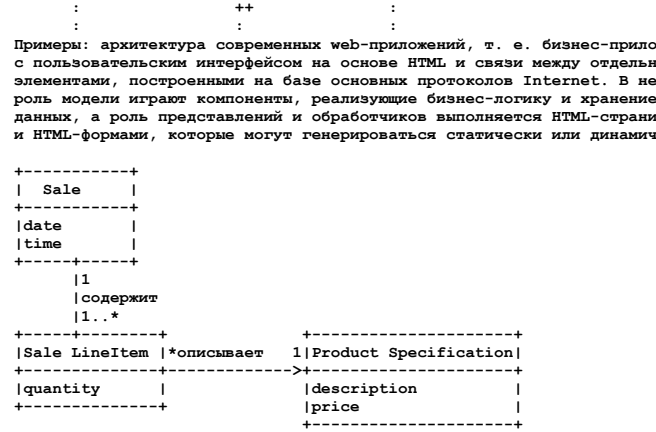
Компонент-модель моделирует данные приложения, реализует основные операции над ними и возможность регистрировать зависимые от него обработчики (Subscriber) и представления. При изменениях данных модель оповещает о них все зарегистрированные компоненты.

Инициализация компонентов:

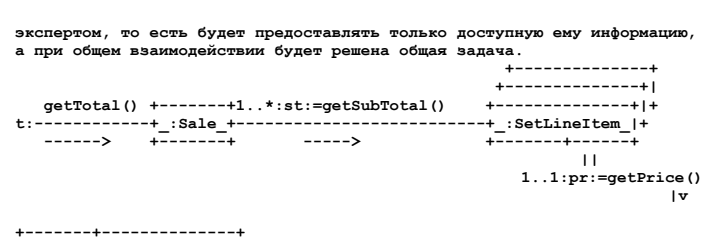


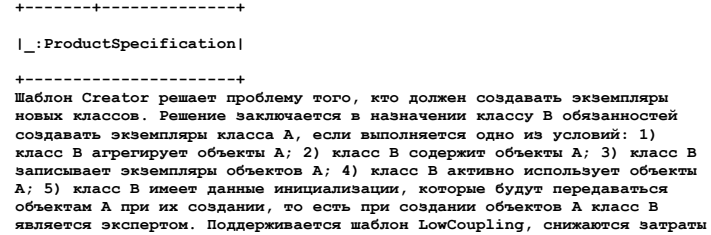
Сценарий обработки определённого действия пользователя с изменения данных и обновления соответствующих им представлений обработчика.

Примеры: архитектура современных web-приложений, т. е. бизнес-приложений с пользовательским интерфейсом на основе HTML и связи между отдельными элементами, построенными на базе основных протоколов Internet. В ней роль модели играют компоненты, реализующие бизнес-логику и хранение данных, а роль представлений и обработчиков выполняется HTML-страницами и HTML-формами, которые могут генерироваться статически или динамически.

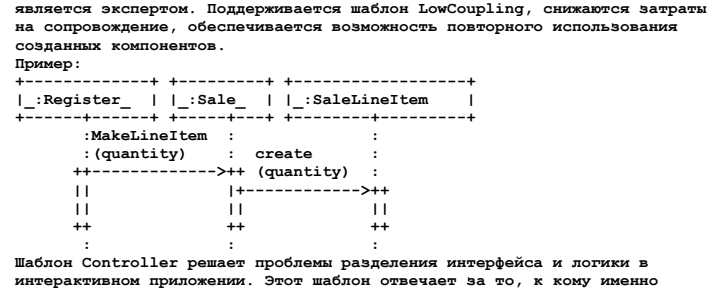


Шаблоны проектирования (design patterns)
GRASP (General Responsibility Assignment Software Patterns) - основные шаблоны распределение обязанностей в программном обеспечении, шаблоны, описывающие фундаментальные принципы распределения обязанностей. Под шаблоном в данном контексте понимают именованную пару "проблема - решение", которая содержит рекомендации для применения в различных конкретных ситуациях. Шаблоны GRASP относятся к этапу проектирования и отвечают за взаимосвязь объектов в системе.
Шаблон Information Expert решает проблему распределения обязанностей между объектами в объектно-ориентированной системе. Под обязанностью в контексте GRASP понимают определённые действия и функцию объекта. Таким образом, Information Expert даёт рекомендации, какие функции должен выполнять тот или иной объект. Решение - назначать обязанность следует информационному эксперту, то есть классу, у которого есть информация, необходимая для выполнения этой обязанности. Не всегда бывает так, что информация содержится в одном классе, чаще она распределена между различными классами, тогда каждый из этих классов будет частичным экспертом, то есть будет предоставлять только доступную ему информацию, а при общем взаимодействии будет решена общая задача.

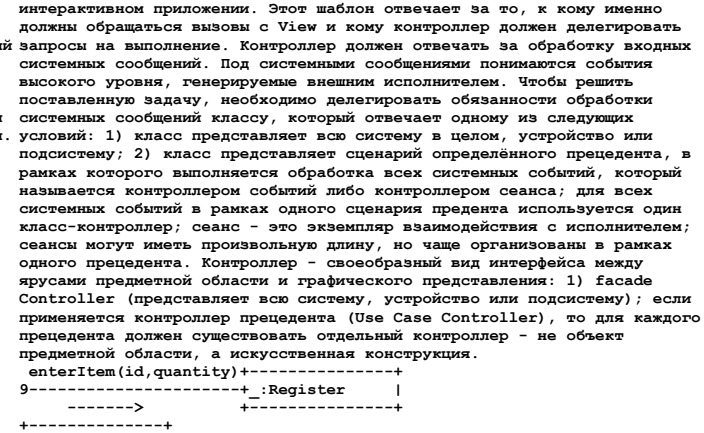


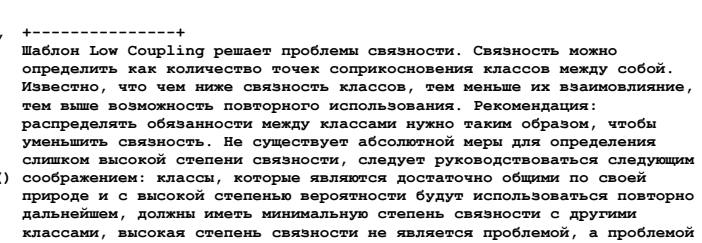


Шаблон Creator решает проблему того, кто должен создавать экземпляры новых классов. Решение заключается в назначении классу B обязанностей создавать экземпляры класса A, если выполняется одно из условий: 1) класс B агрегирует объекты A; 2) класс B содержит объекты A; 3) класс B записывает экземпляры объектов A; 4) класс B активно использует объекты A; 5) класс B имеет данные инициализации, которые будут передаваться объектам A при их создании, то есть при создании объектов A класс B является экспертом. Поддерживается шаблон LowCoupling, снижаются затраты на сопровождение, обеспечивается возможность повторного использования созданных компонентов.
Пример:

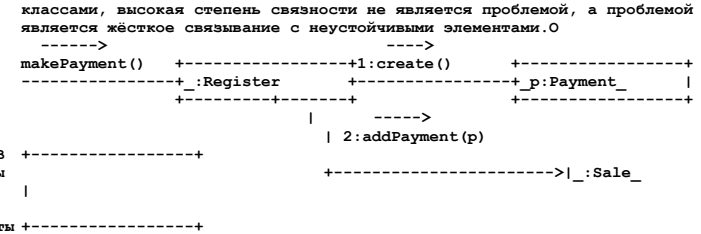


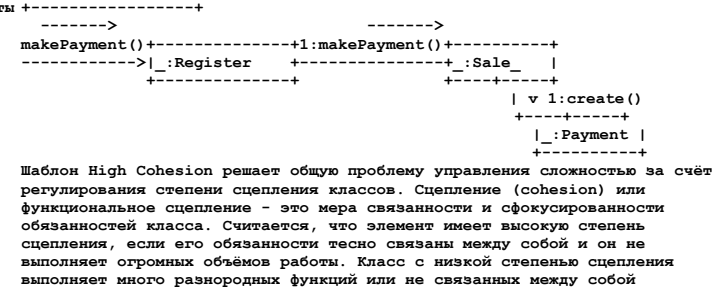
Шаблон Controller решает проблемы разделения интерфейса и логики в интерактивном приложении. Этот шаблон отвечает за то, к кому именно должны обращаться вызовы с View и кому контроллер должен делегировать запросы на выполнение. Контроллер должен отвечать за обработку входных системных сообщений. Под системными сообщениями понимаются события высокого уровня, генерируемые внешним исполнителем. Чтобы решить поставленную задачу, необходимо делегировать обязанности обработки системных сообщений классу, который отвечает одному из следующих условий: 1) класс представляет всю систему в целом, устройство или подсистему; 2) класс представляет сценарий определённого прецедента, в рамках которого выполняется обработка всех системных событий, который называется контроллером событий либо контроллером сеанса; для всех системных событий в рамках одного сценария предента используется один класс-контроллер; сеанс - это экземпляр взаимодействия с исполнителем; сеансы могут иметь произвольную длину, но чаще организованы в рамках одного прецедента. Контроллер - своеобразный вид интерфейса между ярусами предметной области и графического представления: 1) facade Controller (представляет всю систему, устройство или подсистему); если применяется контроллер прецедента (Use Case Controller), то для каждого прецедента должен существовать отдельный контроллер - не объект предметной области, а искусственная конструкция.



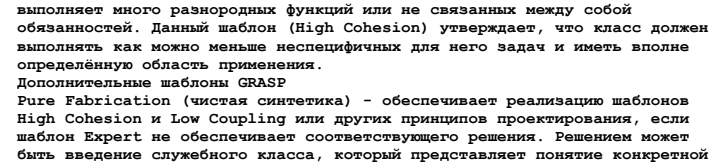


Шаблон Low Coupling решает проблемы связности. Связность можно определить как количество точек соприкосновения классов между собой. Известно, что чем ниже связность классов, тем меньше их взаимовлияние, тем выше возможность повторного использования. Рекомендация: распределять обязанности между классами нужно таким образом, чтобы уменьшить связность. Не существует абсолютной меры для определения слишком высокой степени связности, следует руководствоваться следующим соображением: классы, которые являются достаточно общими по своей природе и с высокой степенью вероятности будут использоваться повторно в дальнейшем, должны иметь минимальную степень связности с другими классами, высокая степень связности не является проблемой, а проблемой является жёсткое связывание с неустойчивыми элементами.O





Шаблон High Cohesion решает общую проблему управления сложностью за счёт регулирования степени сцепления классов. Сцепление (cohesion) или функциональное сцепление - это мера связанности и сфокусированности обязанностей класса. Считается, что элемент имеет высокую степень сцепления, если его обязанности тесно связаны между собой и он не выполняет огромных объёмов работы. Класс с низкой степенью сцепления выполняет много разнородных функций или не связанных между собой обязанностей. Данный шаблон (High Cohesion) утверждает, что класс должен выполнять как можно меньше неспецифичных для него задач и иметь вполне определённую область применения.
Дополнительные шаблоны GRASP
Pure Fabrication (чистая синтетика) - обеспечивает реализацию шаблонов High Cohesion и Low Coupling или других принципов проектирования, если шаблон Expert не обеспечивает соответствующего решения. Решением может быть введение служебного класса, который представляет понятие конкретной предметной области, однако имеет высокую степень сцепления.
Классу Sale необходимо хранить информацию в реляционной базе данных. Задача требует достаточно большого количества специализированных операций, поэтому класс Sale будет иметь низкую степень сцепления. Также класс Sale будет связан с интерфейсом базы данных, что повысит связность, кроме того, задача записи в базу данных является достаточно общей, поэтому необходимо обеспечить повторное использование кода. Решением данной проблемы является создание нового класса, ответственного за сохранение объектов некоторого вида на постоянном носителе (т. е. в базе данных).



Шаблон Indirection (перенаправление) - решает проблему прямой связности, решением является введение промежуточного объекта для обеспечения связи между другими компонентами или службами, которые не связаны между собой напрямую. Классы: Adapter, Facade, Observer. Цель: обеспечить слабую связность за счёт отделения друг от друга основных компонентов.
Шаблон Polymorphysm (полиморфизм) - решает проблему обработки альтернативных вариантов поведения на основе типа, решать эту проблему нужно с использованием полиморфных операций, а не с помощью проверки типа и условной логики. Кроме того, с помощью полиморфизма легко создавать общие компоненты; как результат, получаем следующие преимущества: 1) впоследствии можно легко расширять систему, добавляя новые вариации; 2) новые вариации можно вводить без модификации клиентской части программы.
Шаблон Protected Variations (защищённые вариации) - описывает ключевой принцип, на основе которого реализуются механизмы и шаблоны

программирования и проектирования с целью обеспечения гибкости и защиты системы от влияния внешних систем. Инкапсуляция данных, интерфейсы, полиморфизм, перенаправление - все эти принципы реализуются в рамках данного шаблона. Существует много принципов проектирования, которые являются проявлением этого шаблона, например: 1) проектирование на основе данных; 2) поиск служб; 3) проектирование на основе интерпретатора; 4) рефлексивное проектирование или проектирование на метауровне; 5) унифицированный доступ. Неизменным остаётся базовый принцип шаблона - обеспечивать устойчивость интерфейса.

Надёжность программного обеспечения

Методы увеличения надёжности

Основными методами увеличения надёжности являются: 1) предотвращение ошибок; 2) определение погрешности ошибок. Рекомендации для снижения вероятности возникновения ошибок: 1) не использовать методы с большой вероятностью ошибок (напр., использование указателей); 2) использовать принцип ограниченного доступа, т. е. инкапсуляция, разделение памяти и т. д.; 3) использовать языки-компиляторы с проверкой соответствия типов; 4) использовать языки высокого уровня; 5) строго определять интерфейс; 6) уделять внимание исключениям, таким как пустые множества, пустые циклы, нулевые значения, неинициированные переменные и пр. методы; 7) использовать готовые компоненты; 8) минимизировать различия между моделью и реализацией.

Наиболее опасные техники программирования: 1) использование команды GOTO; 2) использование чисел с плавающей точкой; 3) использование указателей; 4) параллельные вычисления; 5) использование рекуррентных соотношений; 6) использование динамического распределения памяти; 7) процедуры и функции, которые выполняют различные задачи, которые зависят от параметров и внешних условий; 8) использование сложных присвоений без скобок; 9) обработка данных многими процессами без синхронизации в виде блокировок, транзакций и пр.

Базовые принципы кодирования: 1) единые соглашения об именовании переменных; 2) имя переменной должно точно и полностью раскрывать суть того, что она представляет; 3) легче всего отлаживать программу с переменными, которые имеют длину от 10 до 16 символов; 4) при написании программ соблюдаем соглашения, которые являются традиционными для данного языка программирования.

Оформление программы

Цель: точно и последовательно изображать логическую структуру кода, улучшить читабельность программ, оформление должно выдерживать изменения. Хорошее оформление показывает логическую структуру программы.

Форматирование индивидуальных выражений:

1) длина каждого оператора не должна превышать 80 символов;
2) для улучшения читабельности желательно увеличивать количество пробелов;
3) выравнивание группы связанных присваиваний;
4) выравнивание перенесённых строк;
5) использовать одно предложение на строку;
6) избегать посторонних эффектов;
7) форматирование описаний данных;
8) использование одного описания на строку;
9) переменные упорядочиваются по типу;
10) звёздочка пишется возле типа данных;
11) отделение процедур;
12) отделение тела процедуры;
13) модуль программы должен храниться в отдельном файле.

Тестирование программного обеспечения

Ошибки программного обеспечения - это всевозможные несоответствия между демонстрируемыми характеристиками его качества и сформулированными или желательными требованиями и ожиданиями пользователей.

Выделяют 4 типа ошибок:

1) defect - самое общее нарушение любых требований или ожиданий, не обязательно проявляется внешне; к дефектам относятся нарушения стандартов кодирования, недостаточная гибкость системы и т. д.;
2) failure - наблюдаемое нарушение требований, проявляется при некотором реальном сценарии работы программного обеспечения;
3) fault - ошибка в коде программы, которая вызывает нарушение требований при работе, место в программе, которое нужно исправить;
4) error:
   а) ошибка в ментальной модели программиста, которая влечёт за собой ошибки в коде;
   б) некорректные значения данных (выходных или внутренних), которые возникают при ошибках в работе программы.

Основные причины ошибок:

1) неправильное описание задачи;
2) неправильное решение задачи;
3) неправильный перенос решений в код.

Целью тестирования дефектов является обнаружение в программной системе скрытых дефектов до того, как она будет сдана заказчику, при этом важно, что тестирование дефектов демонстрирует их наличие, а не отсутствие в программе.

Тестовые сценарии - это спецификация входных тестовых данных и ожидаемых выходных данных + описание процедуры тестирования.

Альтернативная методика отбора тестовых сценариев базируется на опыте использования подобных систем. В этом случае тестирования подвергаются только определённые средства и функции, работающие с системой.

Тестирование - это проверка соответствия программного обеспечения требованиям, осуществляемая посредством наблюдения за его работой в специальных искусственно построенных ситуациях. Тестирование - конечная процедура, а набор тестовых ситуаций всегда конечен. Набор тестовых ситуаций должен быть настолько мал, чтобы тесты можно было провести в рамках проекта разработки программного обеспечения, не увеличивая значительно бюджет и сроки. Чтобы оценивать качество программного обеспечения посредством тестирования, необходимо иметь критерии полноты тестирования, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними. Этот критерий может утверждать, что всё равно, в какой из ситуаций, A или B, проверять правильность работы программного обеспечения. Часто критерий полноты тестирования задаётся посредством определения эквивалентности ситуаций, которое даёт конечный набор классов ситуаций. Такой критерий называют критерием тестового покрытия, а процент классов [...] достигнутым тестовым покрытием. Основные задачи тестирования - построить такой набор ситуаций, который был бы достаточно представительный и позволял бы завершить тестирование с достаточной степенью уверенности в правильности программного обеспечения вообще и также убедиться, что в конкретной ситуации программное обеспечение работает правильно, в соответствии с требованиями.

Виды тестирования

Тестирования "чёрного ящика" нацелено на проверку требований. Систему при этом рассматривают как чёрный ящик, поведение которого можно определить с помощью изучения его входных и соответствующих выходных данных. Таким образом, тесты для него и критерии полноты тестирования строятся на основе требований и ограничений, чётко зафиксированных в спецификациях, стандартах, внутренних нормативных документах. Часто такое тестирование называется тестированием на соответствие (conformance testing). Его частным случаем функциональное тестирование, при котором испытатель проверяет не реализацию программного обеспечения, а лишь его выполняемые функции. Тесты для него, а также используемые критерии полноты тестирования определяют на основе требований к функциональности.

Входные данные программ часто можно разбить на несколько классов. К одному классу относятся данные, которые имеют общие свойства: положительные числа, отрицательные числа, строки. Обычно для всех данных какого-либо класса поведение программы одинаково и эквивалентно, соответственно, возникает понятие области эквивалентности. Область эквивалентности входных данных - это множество входных данных, все элементы которого обрабатываются одинаково. Области эквивалентности выходных данных - это множества выходных данных, имеющие общие свойства, которые позволяют считать их отдельным классом. Один из систематических методов выявления дефектов заключается в определении всех областей эквивалентности, обрабатываемых программой. Контрольные тесты разбиваются так, чтоб входные и выходные данные лежали в пределах этих областей. После определения областей эквивалентности для каждой из них подбираются тестовые данные, при этом руководствуются следующим правилом: для тестов выбираются данные, которые расположены на границе области эквивалентности, и отдельно данные, которые лежат внутри этой области. Основная причина такого выбора данных состоит в том, что граничные значения часто нетипичны и поэтому игнорируются программистами, хотя зачастую ошибки в программе возникают при обработке подобных нетипичных значений. Ещё одним примером тестирования на соответствие является аттестационное или квалификационное тестирование, по результатам которого программная система получает или не получает официальный документ, подтверждающий её соответствие определённым требованиям и стандартам.

Тестирование "белого ящика" (структурное тестирование) - тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Структурное тестирование применяется в отношении небольших программных элементов, например, к подпрограммам или методам, ассоциированным с объектами. При таком подходе испытатель анализирует программный код и для получения тестовых данных использует знания о структуре компонентов. Например, по анализу кода можно определить, сколько контрольных тестов необходимо выполнить для того, чтобы в процессе тестирования все операторы исполнились хотя бы один раз. Знание алгоритма, используемого при реализации некоторой функции, можно применять для определения областей эквивалентности. Критерии полноты тестирования основаны на проценте элементов кода, которые отработали в ходе выполнения тестов.

Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определёнными ограничениями на значения внутренних данных системы, например, значения параметров, вызовов, результатов локальных переменных и прочие ограничения.

Метод тестирования ветвей - метод структурного тестирования, при котором проверяются все независимо выполняемые ветви приложения или компонента. Если выполняются все независимые ветви, то и все операторы должны выполняться по крайней мере один раз. Все условные операторы тестируются как с верными, так и с ложными значениями условий. В объектно-ориентированных системах данный метод используется для тестирования методов, ассоциированных с объектами. Количество ветвей в программе обычно пропорционально её размеру. Методы тестирования ветвей, как правило, используются для тестирования отдельных программных элементов и модулей.

Тестирование, нацеленное на определённые ошибки: целью является выявление определённых ошибок, полнота тестирования выясняется на основе отношения количества проверенных ситуаций относительно общего числа ситуаций. Тестирование "на отказ" (smoke testing), в ходе которого пытаются вывести систему из строя, давая ей на вход как обычные данные, так и некорректные, с намеренно внесёнными ошибками. Другим примером служит метод оценки полноты тестирования с помощью набора мутантов, т. е. программ, которые совпадают с тестируемой всюду, кроме нескольких мест, где специально внесены определённые ошибки.Таким образом, чем больше мутантов не проходит успешно через данный набор тестов, тем полнее и качественнее проведено тестирование.

Стратегии тестирования

Виды тестирования исходя из уровня, на которые оно нацелено:

1) модульное тестирование (unit testing) - предназначено для проверки правильности отдельных модулей независимо от их окружения. При этом проверяется, что если модуль получает на вход данные, определяющие определённых критериям корректности, то и результаты его корректны. Для описания критериев корректности входных и выходных данных часто используют программные контракты, то есть предпосылки, которые описывают для каждой операции, на каких входных данных она должна работать, также постусловия, которые описывают для каждой операции, как должны соотноситься входные данные с возвращаемыми ей результатами инварианты, определяющие критерий целостности внутренних данных модуля. Модульное тестирование - составная часть налаживаемого тестирования и выполняется на этапе отладки;
2) интеграционное тестирование (integration testing) - предназначено для проверки правильности взаимодействия модулей определённого набора друг с другом, при этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предпосылок вызываемых операций. Данная стратегия используется также при отладке на более позднем этапе разработки;
3) системное тестирование (system testing) - предназначено для проверки правильности работы системы в целом, её способности правильно решать поставленные пользователем задачи в различных ситуациях.

Системное тестирование выполняется через внешние интерфейсы программного обеспечения и тесно связано с тестированием интерфейсов, которое проводят с помощью имитации действий пользователей над элементами этого интерфейса. Частным случаем этого вида тестирования является тестирование графического интерфейса пользователя (GUI) и интерфейса Web-приложений. Если интеграционное и модульное тестирование чаще проводят, воздействуя на компоненты системы с помощью предоставляемого ими программного интерфейса (API), то на системном уровне не обойтись без интерфейса пользователя, хотя тестирование через API также возможно.

Регрессивное тестирование - это методика испытаний, при которой тесты, произведенные ранее, повторно выполняются на новой версии целевого объекта. Цель такого тестирования - обеспечить, чтоб качество целевого объекта не ухудшалось при добавлении к этому объекту новых функций. Регрессивное тестирование необходимо для максимально раннего выявления дефектов. Как правило, такое тестирование выполняется в каждой итерации и заключается в повторном запуске тестов, выполненных на предыдущих итерациях.

Основные принципы дизайна классов в объектно-ориентированном программировании (SOLID) (предложены Робертом Мартином): 1) Single Responsibility Principle (принцип единственной обязанности): не должно быть более одной причины для изменения класса; 2) Open/Closed Principle - принцип открытия/закрытия: объекты проектирования (классы, функции, модули) должны быть открыты для расширения, но закрыты для модификации; открыт для расширения - поведение может быть расширено путём добавления новых объектов, реализующих новые аспекты поведения; закрыт для модификации - в результате расширения поведения исходный или двоичный код объекта не может быть изменён; 3) Liskov Substituion Principle (принцип замещения Лисков): функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты произвольных классов, не зная об этом; 4) Interface Segregation Principle (принцип разделения интерфейса): клиент не должен вынужденно зависеть от элементов интерфейса, которые он не использует (зависимость

между классами должна быть ограничена как можно более узким интерфейсом; 5) Dependency Invertion Principle (принцип обращения зависимостей: модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций, а абстракции не должны зависеть от деталей, при этом детали должны зависеть от абстракций.

Модернизация программного обеспечения

Динамика развития программ

Под динамикой развития программ понимают исследование изменений в программной системе. Результатом этих исследований стало множество законов Лемана, касающихся модернизации систем: 1) непрерывность модернизации: для программ, эксплутируемых в реальных условиях, модернизация - это необходимость, иначе их полезность снижается; 2) растущая сложность: согласно развитию, программы становятся всё более сложными, для упрощения или сохранения их структуры необходимы дополнительные затраты; 3) эволюция больших систем: процесс развития систем - саморегулирующийся. Такие характеристики системы, как размер, время между выпусками очередных версий и количество регистрируемых ошибок для каждой версии остаются практически неизменными; 4) организационная стабильность: жизненный цикл системы относительно стабилен, независимо от средств, которые выделены или не выделены на её развитие; 5) стабильность количества изменений: за весь жизненный цикл системы количество изменений в каждой версии остаётся примерно одинаковым.

Сопровождение программного обеспечения

Сопровождение - это обычный процесс изменения системы после её поставки заказчику, сопровождение не связано со значительными изменениями архитектуры системы, изменение существующих компонентов системы или добавление новых. Выделяют три вида сопровождения системы: 1) сопровождение с целью исправления ошибок: ошибки программирования легко одолеть, если же возникают ошибки проектирования, они требуют корректировки и перепрограммирования некоторых компонентов; 2) сопровождение с целью адаптации программного обеспечения к специфическим условиям эксплуатации; это может понадобиться при изменении составляющих рабочего окружения системы (программных средств, операционной системы); 3) сопровождение с целью изменения функциональных возможностей системы; возникает в результате организационных или деловых изменений.

Ключевые факторы, определяющие стоимость разработки и сопровождения: 1) стабильность команды разработчиков; 2) ответственность в соответствии с контрактом; 3) квалификация специалистов; 4) возраст и структура программы.

Показатели для оценки удобства сопровождения после ввода системы в эксплуатацию: 1) количество запросов на корректировку системы: рост количества сообщений о сбоях в системе означает увеличения количества ошибок, которые подлежат исправлению при сопровождении; это говорит об ухудшении удобства сопровождения; 2) среднее время, затраченное на анализ причин системных сбоев и отказов: этот показатель пропорционален количеству системных компонентов, в которых нужно внести изменения; если этот показатель растёт, то система требует многочисленных изменений; 3) среднее время, необходимое для реализации изменений: учитывается не время анализа системы для выявления причин сбоев, а время реализации изменений, их документирование, которое зависит от сложности программного кода; увеличение этого показателя означает сложность сопровождения; 4) количество незавершённых запросов на изменение: с ростом количества таких запросов сопровождение системы становится труднее. Таким образом, для определения стоимости сопровождения используется дополнительная информация о запросах на изменение и прогнозирование по удобству сопровождения. Причины перехода к новой архитектуре системы: 1) стоимость аппаратных средств; 2) совершенствование интерфейса пользователя; 3) распределённый доступ к системам.

Модели. Процессы разработки программного обеспечения

Жизненный цикл программной системы

На протяжении жизненного цикла программное обеспечение проходит через: 1) анализ предметной области; 2) сбор требований; 3) проектирование; 4) кодирование; 5) тестирование; 6) сопровождение и другие виды деятельности.

Модель жизненного цикла программного обеспечения описывает набор фаз (этапов, стадий) проекта по созданию программного обеспечения, на которых выполняются отдельные процессы, разбитые на операции и задачи.

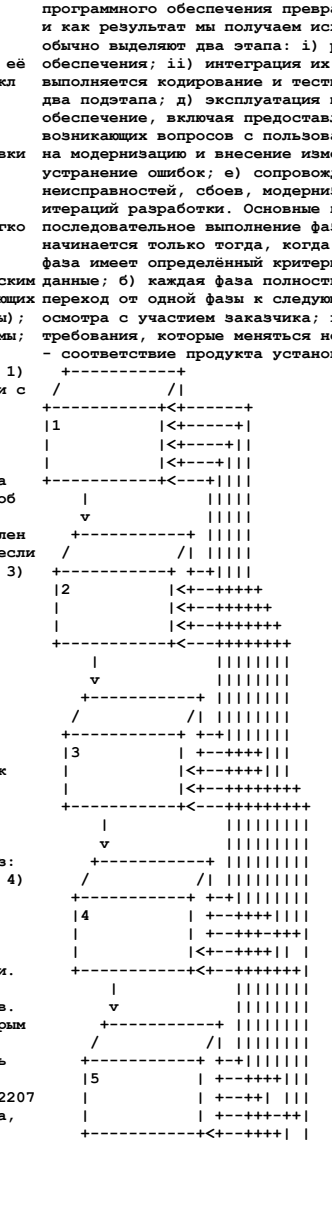
Фаза проекта - это объединение логически связанных операций проекта, которые обычно завершаются достижением одного из основных результатов.

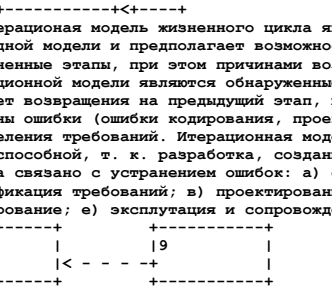
Процесс - это набор взаимосвязанных ресурсов и работ, благодаря которым входные воздействия преобразуются в выходные результаты.

Операция - это элемент работ проекта, при этом у операции обычно есть ожидаемая продолжительность, потребность в ресурсах и стоимость.

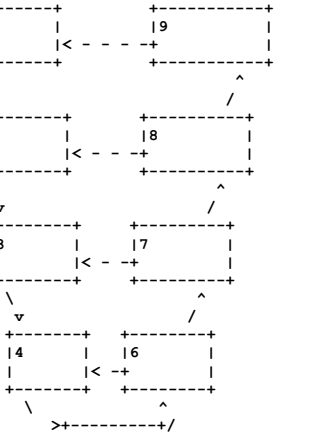
Операции могут подраспределяться на задачи: согласно стандарту ISO 12207 понятие жизненного цикла (lifecycle model) определяется как структура, состоящая из процессов, работ и задач, содержащих разработку, эксплуатацию и сопровождение программного продукта и охватывающее жизнь системы от установления требований к ней до прекращения её использования, при этом конкретные модели определяются особенностью задач, ограничением на ресурсы, опытом разработчиков и прочими факторами.

Рациональные модели процесса разработки: 1) каскадная модель (waterfall) предусматривает последовательное выполнение следующих фаз: а) исследование концепции: на этой фазе происходит исследование требований, разрабатывается виденье продукта и оценивается возможность его реализации; б) определение требований: определяются программные требования для информационной предметной области системы, назначение линии поведения, а также производительность интерфейса; в) разработка проекта: разрабатывается и формулируется логически последовательная техническая характеристика программной системы, включая структуры данных, архитектуру программного обеспечения, предоставление интерфейса и процессуальную детализацию; г) реализация: эскизное описание программного обеспечения превращается в полноценный программный продукт и как результат мы получаем исходный код, базу данных и документацию; обычно выделяют два этапа: i) реализация компонент программного обеспечения; ii) интеграция их в готовый продукт; на обоих этих этапах выполняется кодирование и тестирование, и порой они рассматриваются как два подэтапа; д) эксплуатация и поддержка: 1) запуск и текущее обеспечение, включая предоставление технической помощи. обсуждение возникающих вопросов с пользователем, регистрация запросов пользователя на модернизацию и внесение изменений, а также корректировки или устранение ошибок; е) сопровождение: устранение программных ошибок, неисправностей, сбоев, модернизация, внесение изменений. Фаза состоит из итераций разработки. Основные принципы каскадной модели: а) строго последовательное выполнение фаз, а именно: каждая следующая фаза начинается только тогда, когда закончится предыдущая, при этом каждая фаза имеет определённый критерий входа и выхода, входные и выходные данные; б) каждая фаза полностью документируется; в) документируется переход от одной фазы к следующей, осуществляется с помощью формального осмотра с участием заказчика; г) основа модели: сформулированы требования, которые меняться не должны; д) критерий качества результата - соответствие продукта установленным требованиям.

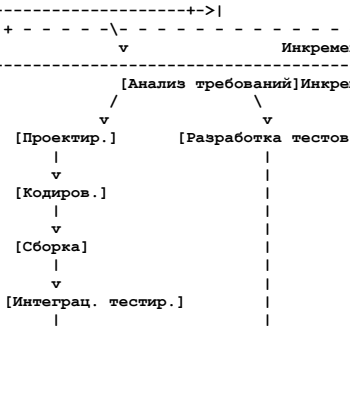


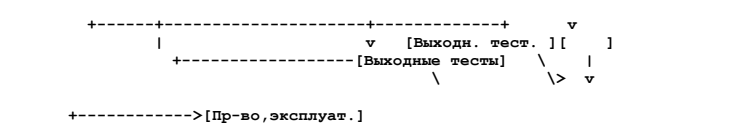


2) итерационая модель жизненного цикла является развитием классической каскадной модели и предполагает возможность возврата на ранее выполненные этапы, при этом причинами возврата в классической итерационной модели являются обнаруженные ошибки, устранение которых требует возвращения на предыдущий этап, в зависимости от типа ошибки и причины ошибки (ошибки кодирования, проектирования, спецификации или определения требований. Итерационная модель является более жизнеспособной, т. к. разработка, создание программного обеспечения всегда связано с устранением ошибок: а) определение требований; б) спецификация требований; в) проектирование; г) реализация; д) тестирование; е) эксплуатация и сопровождение;



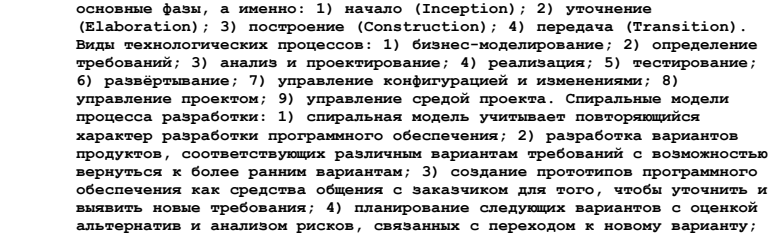
3) V-образная модель: создана как итерационная разновидность каскадной модели: в) высокоуровневое проектирвоание; г) детальное проектирование; д) кодирование; е) модульное тестирование; ё) сборка и тестирование; ж) системное тестирование; з) производство и эксплуатация; целями итераций в этой модели является обеспечение процесса тестирования, при этом тестирование продукта обсуждается, проектируется и планируется на ранних этапах жизненного цикла разработки. План испытания и приёмки заказчиком разрабатывается на этапе планирования, а компоновочного испытания системы - на фазах анализа, разработки проекта и т. д.; процесс разработки планов испытания обозначен пунктирной линией, и кроме планов на ранних этапах разрабатываются также и сами тесты, которые будут выполняться при завершении параллельных этапов;



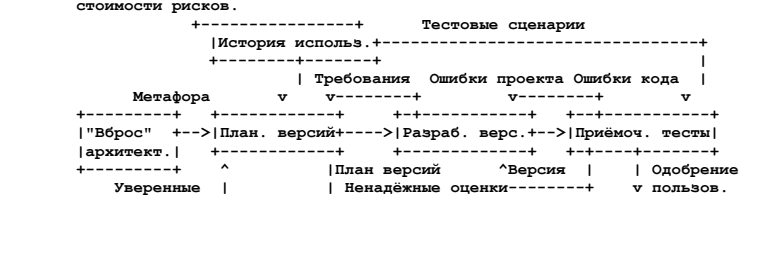


+------------>[Пр-во,эксплуат.]
4) инкрементная разработка: является процессом поэтапной реализации всей системы и поэтапного наращивания (увеличения) функциональных возможностей: а) необходим полный, заранее сформулированный набор требований, которые делятся по определённому признаку на части; б) выбирается первая группа требований и выполняется полный проход по каскадной модели; в) после того, как первый вариант системы, который выполняет первую группу требований, сдан заказчику, разработчики переходят к следующему шагу, которые выполняет вторую группу требваний, и т. д.
Д/з: модель RUI

Модель RUI является итерационно-инкрементной и в ней выделяют четыре основные фазы, а именно: 1) начало (Inception); 2) уточнение (Elaboration); 3) построение (Construction); 4) передача (Transition).
Виды технологических процессов: 1) бизнес-моделирование; 2) определение требований; 3) анализ и проектирование; 4) реализация; 5) тестирование; 6) развёртывание; 7) управление конфигурацией и изменениями; 8) управление проектом; 9) управление средой проекта. Спиральные модели процесса разработки: 1) спиральная модель учитывает повторяющийся характер разработки программного обеспечения; 2) разработка вариантов продуктов, соответствующих различным вариантам требований с возможностью вернуться к более ранним вариантам; 3) создание прототипов программного обеспечения как средства общения с заказчиком для того, чтобы уточнить и выявить новые требования; 4) планирование следующих вариантов с оценкой альтернатив и анализом рисков, связанных с переходом к новому варианту; 5) возможность перехода к разработке следующего варианта до завершения предыдущего в случае, когда риск завершения очередного варианта становится неоправданно высок; 6) использование каскадной модели как схемы разработки очередного варианта; 7) активное привлечение заказчика к работе над проектом: заказчик принимает участие в оценке очередного прототипа программного обеспечения, уточнении требований при переходе к следующему, оценке предложенных альтернатив очередного варианта и оценке рисков.



2. Оценка альтернатив, выделение рисков и способов борьбы с ними.
AR - рабочий прототип
3. Разработка следующего уровня.
4. Планирование следующих фаз разработки.
4.1. План жизненного цикла и работы с требованиями.
3.3. Проверка требований.
4.2. План разработки.
3.4. Проект программного обеспечения.
3.5. Проверка проекта.
4.3. План интеграции, план тестирования, сборка.
4.4. Рабочий прототип.
3.6. Детальный проект.
3.7. Кодирование.
3.8. Модульное тестирование.
3.9. Сборка.
3.10. Внедрение.
Дополнительные особенности: 1) до начала разработки есть несколько циклов анализ требований и проектирования; 2) количество циклов модели не ограничено и зависит от сложности и объёма задачи; 3) в модели предусмотрены возвращения к поставленным вариантам при изменении стоимости рисков.

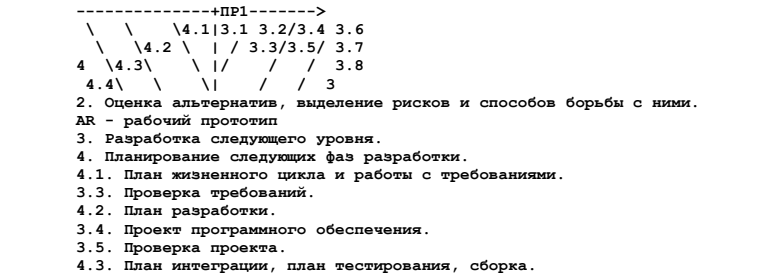


оценки |+--------+|
++"Вброс" |<
|решения |
| |
+--------+

Итерация +----------+
|Финальн. |
|версия |
|небольш. |
+----------+

Microsoft Solution Framework
Тесная связь с заказчиком, требования заказчика - это проявление неких проблем. MSF уделяет большое внимание анализу проблем заказчика и разработке вариантов системы для поиска решений этих проблем. Модель жизненного цикла MSF является гибридом каскадной и спиральной и ориентирована на т. н. "вехи" (milestones), характеризующие достижение определённого существенного результата. Нужно ответить на вопрос, достигли ли мы цели на данном этапе.

Developing\<>/-
Создание общей картины
Планирование
1. Утверждение документа общей картины
Разработка
Стабилизация
2. Утверждение проектных планов
3. Окончательное утверждение области действия проекта
4. Подтверждение готовности проекта к выпуску
Развёртывание
5. Развёрнутое решение
Использование эволюционных моделей процесса разработки



2) быстрый анализ
3) создание базы данных
4) интерфейс пользователя
5) функции
6) итерационное прототипирование
7) утверждение заказчика
8) производная разработка проекта
9) подгонка
10) эксплуатация и сопровождение
Модель быстрого протипирования нужна для быстрого создания прототипов продуктов с целью уточнения требований и поэтапного развития протипов до конечного продукта. Высокая производительность обеспечивается планированием разработки прототипов и участием заказчика в процессе разработки.
Agile Development Method (метод живой разработки) - используется во многих компаниях по производству программного обеспечения, примером его использования является экстремальное программирование (Extreme Programming). Модель жизненного цикла XP является итерационно-инкрементной моделью быстрого создания и модификации прототипа продукта, который удовлетворяет очередному требованию. В основе этого метода лежат т. н. user story (истории использовании)
Управление программными проектами
Продукт - это усилие ... Свойства проекта - уникальность, ограниченность во времени, ограниченность ресурсов, сложность, неопределённость, предсказуемость. Project Management (управление проектами) - это наука и искусство управления и координации человеческих и материальных ресурсов на протяжении жизненного цикла проекта за счёт применения современных методов и техники управления для достижения определённых результатов по составу, объёму работ, стоимости, времени и качеству, которые будут удовлетворять участников этого проекта. Задачи: 1) управление содержанием проекта и качеством; 2) управление ресурсами проекта; 3) управление рисками; 4) управление коммуникациями и информационное обеспечение проекта; 5) управление конфигурациями и изменениями; 6) управление проектной средой и технологиями; 7) контроль и мониторинг состояния проекта.
Планирование и мониторинг работ
Планирование является неким регламентом или соглашением о том, как предполагается достигнуть поставленную цель.
Основные функции планирования:
1) преобразование потребностей на управляемые задачи;
2) определение необходимых ресурсов;
3) координация командной работы над проектом;
4) оценка потенциальных рисков;
5) сигнализация о возникающих проблемах.
В ходе ответа на вопрос определяются цели проекта, стратегии достижения целей, выделяются задачи.
Как нужно это сделать?
В результате составляется график выполнения отдельных задач, планируется бюджет, решается, кто будет выполнять ту или иную стадию проекта, выполняется планирование человеческих ресурсов, распределение ответственности между исполнителями, определение качественных характеристик (планирование качества), планирование рисков (определяются возможные помехи в процессе выполнения проекта), формулировка метрик проекта, разбиение единого проекта на задачи и подзадачи.
Структурная декомпозиция работ (Work Breakdown Structure) - иерархическая декомпозиция и организация деятельностей и подзадач, действий для удовлетворения целей проекта.
Контроль - это текущая деятельность, осуществляемая для того, чтобы компенсировать неизбежное отклонение от плана, при этом отдельно могут контролироваться как общий ход выполнения проекта, контроль общих видов работ, контроль отдельных исполнителей и пр. Для того, чтобы выполнять контроль, необходимо вводить метрики - количественные показатели, оценки различных характеристик проекта и процесса его выполнения.
Управление рисками
В ходе выполнения процесса могут возникать некие угрозы, поэтому необходимо анализировать и планировать их возникновение, при этом те неопределённости, которые возникают в положительном смысле - это благоприятные возможности, если в отрицательном - это риски проекта.
Классификация рисков:
1) риски проекта, которые влияют на его течение:
1.1) технологические риски, которые связаны с производительностью технологий, инструментов, которые вовлечены в процесс;
1.2) кадровые риски: конфликтные ситуации, возникающие в команде, несоответствие квалификации, отсутствие необходимых навыков для выполнения тех или иных задач, недостаточная производительность;
1.3) риски требований;
1.4) коммерческие риски: всё, что связано с оценкой расходов проекта, неверной оценкой рынков сбыта;
1.5) управленческие риски;
1.6) производственные риски;
2) риски, влияющие на результаты проекта:
2.1) технические риски;
2.2) качество результатов проекта;
2.3) эксплуатационные риски;
3) бизнес-риски:
3.1) контрактные риски: всё, что связано с соисполнителями проектов;
3.2) инвестиционные риски: финансирование со стороны инвесторов, их финансовое вливание в Ваш проект;
3.3) риски сбыта: этапы сбыта Вашего программного продукта;
3.4) конънктурные риски.
Управление риском - это комплекс мероприятий по выявлению, оценке, предотвращению и контролю рисков проекта. Оценка состоит из определения, анализа и назначения приоритетов. Контроль состоит из планирования управления, решений, мониторинга и управления качеством (в международных стандартах сформулировано понятие "качество программного продукта", также качество - это соответствие его потребностям заказчика).
Требования - неадекватность функционирования программного продукта, недостаточное взаимодействие программного продукта с другими программами, а также аппаратными и коммуникационными средствами, отказы программного продукта в процессе использования по назначению (ISO 9000), замедленное время работы программного обеспечения и задержка предоставления промежуточной выходной информации (ISO 12207), неполнота отображения информации, несоответствие хранимых данных информации тем, которые вводятся оператором (TickIT, SEI), потеря актуальности информации, хранимой в системе (SW-CMM, Trilium), нарушение конфиденциальности информации (ISO 15504), содержание сопроводительной документации и справочной системы (CMMI).