ТП – 2019

Определения

Декларативное программирование – это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. HTML.

Аспектно-ориентированное программирование – парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули.

Метод – последовательный процесс создания ряда моделей, которые описывают различные стороны ПС вполне определёнными средствами. (Г.Буч)

Методология определяет руководящие указания в процессе разработки ПС. (Г.Буч)

ТП (технология разработки ПО) – наука об оптимальных способах (приёмах) проведения процесса программирования.

Программа – последовательность команд (или операторов), которая после декодирования её ВМ и транслятором выполняет некоторую работу. (Фокс)

ПО – группа взаимосвязанных и взаимодействующих модулей или программ.

Программный продукт – программа (или ПО), которой желающий может воспользоваться, должна быть надёжной и эффективной.

Программное изделие – программный продукт, который удовлетворяет определённому стандарту.

Программная система – программный продукт + что-то ещё (например, соответствующая платформа, ОС, сервисное ПО и др.)

Пакет прикладных программ (ППП) – программный комплекс, который предназначен для автоматизации решения определённого класса задач.

Модуль – выделенная (по тем или иным мотивам) часть первичного программного продукта. (Майерс). Модульный принцип является общепринятым в практике программирования, т.е. является обязательным для любой серьёзной программной разработки.

Тестирование программного обеспечения (software testing) – это процесс анализа или эксплуатации программного обеспечения с целью выявления дефектов.

1. История развития языков программирования высокого уровня.

Основные тенденции развития языков программирования:

1. Перемещение акцентов от программирования отдельных деталей к программированию более крупных компонент.
2. Развитие и совершенствование языков программирования *высокого уровня*.
3. Коммерческие программные системы становятся больше и сложнее → проводятся прикладные исследования по методологии проектирования (особенно по декомпозиции, абстрагированию и иерархиям), создание более выразительных языков программирования.
4. Переход от *императивных* (указывают компьютеру, что делать) к *декларативным* языкам (описывают ключевые абстракции проблемной области).

Вэгнер сгруппировал наиболее известные языки в четыре поколения в зависимости от языковых конструкций, которые впервые в них появились. В каждом поколении менялись поддерживаемые языками механизмы абстракции. Каждое поколение приближало программирование к предметной области и удаляло от конкретной машины.

1-е поколение (1954-1958): FORTRAN I, ALGOL-58, Flowmatic, IPL V (математические формулы). Ориентировались на научно-инженерные применения, словарь предметной области почти исключительно математический. Созданы для ***упрощения программирования математических формул***, чтобы освободить программиста от ассемблера и машинного кода.

2-е поколение (1959-1961): FORTRAN II (подпрограммы, раздельная компиляция), ALGOL-60 (блочная структура, типы данных), COBOL (описание данных, работа с файлами), Lisp (обработка списков, указатели, сборка мусора). Развитие ***алгоритмических абстракций***. Мощность компьютеров росла, расширялась область их применения (бизнес). Главная задача – инструктировать машину, что делать (прочти анкеты, отсортируй, выведи результаты).

3-е поколение (1962-1970): PL/I (FORTRAN+ALGOL+COBOL), ALGOL-68 (более строгий преемник ALGOL-60), Pascal (более простой преемник ALGOL-60), Simula (классы, абстрактные данные). Появились транзисторы и интегральные схемы → стоимость компьютеров снизилась, производительность росла. Появилась возможность решать более сложные задачи, но для этого требуется обрабатывать разнообразные типы данных. Языки стали поддерживать ***абстракцию данных***, программисты могут описывать свои собственные типы данных.

Потерянное поколение (1970-1980): много языков создано, мало выжило. Создано ~2000 языков и их диалектов. Новые языки должны быть приспособлены для написания ***крупных программных систем***. Языки (Smalltalk, Ada, CLOS, C++, Eiffel) созданы на основе более ранних языков (Simula, ALGOL-68, Lisp, C).

Наибольший интерес представляют *объектные* (нет наследования) и *объектно-ориентированные* (есть наследование) языки, которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции программного обеспечения.

2. Архитектура языков программирования (три поколения).

Под топологией будем иметь в виду основные элементы языка программирования и их взаимодействие.

1-е и начало 2-го поколения. Основным строительным блоком является *подпрограмма*. Программы имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм. ***Проблемы:***

- нельзя явно разделить разнотипные данные;
- область данных открыта всем подпрограммам → ошибка в какой-либо части программы может иметь далеко идущие последствия, трудно гарантировать целостность данных;
- в процессе эксплуатации возникает путаница из-за связей между подпрограммами, запутанных схем управления, неясного смысла данных → угроза надежности системы, неясность программы.

Конец 2-го и начало 3-го поколения. Развитие предыдущего поколения, но не решены проблемы программирования "в большом" (=разработка больших проектов) и проектирования данных. Осознается важность подпрограмм как механизма абстрагирования. ***Последствия:***

- разработаны языки, поддерживающие механизмы *передачи параметров*;
- заложены основания *структурного программирования* (**механизмы вложенности подпрограмм**, исследование структур управления и областей видимости);
- возникли методы *структурного проектирования*, стимулирующие разработчиков создавать большие системы, используя подпрограммы как готовые строительные блоки.

Конец 3-го поколения. Для программирования "в большом" начал развиваться новый механизм структурирования: *модульное программирование*. Программные проекты разрастались → большие коллективы программистов → независимая разработка отдельных частей проекта. Отдельно компилируемый модуль сначала был просто набором данных и подпрограмм, которые, скорее всего, будут изменяться совместно. В большинстве языков этого поколения поддерживалось модульное программирование, но не было правил, обеспечивающих согласование интерфейсов модулей (в результате ожидаемые и фактические параметры могут не совпадать; модуль может использовать область данных другого модуля и т.д.).

3. Архитектура объектно-ориентированных языков программирования.

Сложность задачи часто обусловлена сложностью объектов, с которыми нужно работать. Поэтому:

1. Возникают методы проектирования на основе потоков данных, которые вносят упорядоченность в абстракцию данных в языках, ориентированных на алгоритмы.
2. Появляется теория типов данных, которая воплощается в Pascal, Simula и последующих языках 1970-1980 гг.: Smalltalk, Object Pascal, C++, CLOS, Ada. Эти языки получили название объектных или объектно-ориентированных.

Особенности:

- основным элементом конструкции является ***модуль***, составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения;
- структура программ представляется ***графом***, а не деревом, как для алгоритмических языков;
- уменьшена или ***отсутствует область глобальных данных***;
- основными логическими строительными блоками систем становятся ***классы и объекты***, а не алгоритмы.

Объектный подход масштабируется и может быть применен на все более высоких уровнях (программирование "в огромном"). Кластеры абстракций в больших системах могут представляться в виде многослойной структуры: внутри каждого кластера можно выделить группы взаимодействующих абстракций.

Объектный подход позволяет справляться со сложностью систем самой разной природы, поэтому он используется не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитектуры компьютеров.

4. Сложность, присущая ПО (основные причины). Проблемы, возникающие при создании сложных систем.

Нас интересует разработка промышленных программных продуктов. Системы подобного типа обычно имеют большое время жизни и большое количество пользователей.

Сложность присуща всем большим программным системам. Причины:

- сложность реальной предметной области, из которой исходит заказ на разработку ("нестыковка" между заказчиками и разработчиками);
- трудность управления процессом разработки (большой коллектив);
- необходимость обеспечить достаточную гибкость программы;
- неудовлетворительные способы описания поведения больших дискретных систем (в реальном мире системы непрерывные).

Проблемы, возникающие при создании сложных систем. Последствия неограниченной сложности: "чем сложнее система, тем легче ее полностью развалить". Кризис ПО: проекты выходят за рамки установленных сроков и бюджетов и не соответствуют начальным требованиям.

5. Структура сложных систем (пять признаков). Примеры сложных систем (выделить признаки).

Пять признаков сложной системы:

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровням.
2. Выбор, какие компоненты в данной системе считаются элементарными, оставляется на усмотрение исследователя.
3. Связь внутри компонента обычно сильнее, чем связь между компонентами.
4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.
5. Любая работающая сложная система является результатом развития работавшей более простой системы (нельзя спроектировать сложную систему "с нуля").

Примеры сложных систем:

Структура персонального компьютера. Большинство ПК состоит из одних и тех же основных элементов: системной платы, монитора, клавиатуры и устройства внешней памяти (гибкого или жесткого диска). Любая из этих частей может быть также разложена на составляющие (системная плата = ОП + ЦП + шина, ЦП = регистры + схемы управления, ..., диоды + транзисторы и т.д.). Это пример сложной иерархической системы. Различные уровни абстракции могут быть рассмотрены отдельно. Каждый уровень абстракции включает в себя набор устройств, которые совместно обеспечивают некоторые функции более высокого уровня. Мы выбираем, какой уровень абстракции рассматривать, исходя из наших потребностей.

Структура растений и животных. Растения – это сложные многоклеточные организмы. Так же, как у компьютера, части растения образуют иерархию. Растение = корни + стебли + листья, корень = отростки + верхушка, …, набор клеток, клетка = хлоропласт + ядро + ... Существует четкое разделение функций различных уровней абстракции. Поведение целого сложнее, чем поведение суммы его составляющих. Многоклеточные животные, как и растения, имеют иерархическую структуру: клетки формируют ткани, ткани работают вместе как органы, группы органов определяют систему (например, пищеварительную) и так далее.

Структура вещества. Также иерархическая структура: галактики объединены в скопления, звезды, планеты и другие небесные тела образуют галактику. Атомы состоят из электронов, протонов и нейтронов; протоны и нейтроны формируются из кварков.

6. Типовая и структурная иерархии в объектно-ориентированной методологии.

Каноническая форма сложной системы. Обнаружение общих абстракций и механизмов значительно облегчает понимание сложных систем. Наиболее интересные сложные системы содержат много разных иерархий. Существует два типа иерархий: **структурная** (“part of”, «быть частью») и **типовая** («is a»), их также называют структурами объектов и классов соответственно.

Объединяя понятия структуры классов и структуры объектов с пятью признаками сложных систем, мы приходим к тому, что фактически все сложные системы можно представить одной и той же (канонической) формой. ***Особенности***:

- каждая иерархия является многоуровневой: классы и объекты более высокого уровня построены из более простых;
- выбор элементарного класса или объекта зависит от рассматриваемой задачи;
- объекты одного уровня имеют четко выраженные связи, особенно это касается компонентов структуры объектов;
- внутри любого рассматриваемого уровня находится следующий уровень сложности;
- структуры классов и объектов не являются независимыми: каждый элемент структуры объектов представляет специфический экземпляр определенного класса;
- объектов в сложной системе обычно гораздо больше, чем классов.

Показывая обе иерархии, мы демонстрируем избыточность рассматриваемой системы (если бы не знали структуру классов, пришлось бы повторять одни и те же сведения для каждого экземпляра класса).

Структуры классов и объектов системы называют ***архитектурой системы***.

7. Основные понятия: метод, методология, технология. Классификация методов проектирования ПС. Общая характеристика методов проектирования.

Метод – это последовательный процесс создания ряда моделей, которые описывают различные стороны ПС вполне определенными средствами. У каждого метода (или группы методов) имеются свои механизмы и нотация для создания и описания модели.

Методология определяет руководящие указания в процессе разработки ПС, т.е. должна определять этапы процесса разработки, выбор методов разработки, указания для оценки качества проекта, и объединяет механизмы, используемые в процессе разработки, одним общим философским подходом.

Классификация методов проектирования (Соммервиль):

1) Методы структурного проектирования (SD). Большое внимание алгоритмической декомпозиции, основной строительный блок – подпрограмма.
2) Методы организации потоков данных (DD). Структура программы строится как формальная организация преобразования входных потоков в выходные.
3) Методы объектно-ориентированного проекта. ПС проектируется как совокупность взаимодействующих объектов – экземпляров класса.

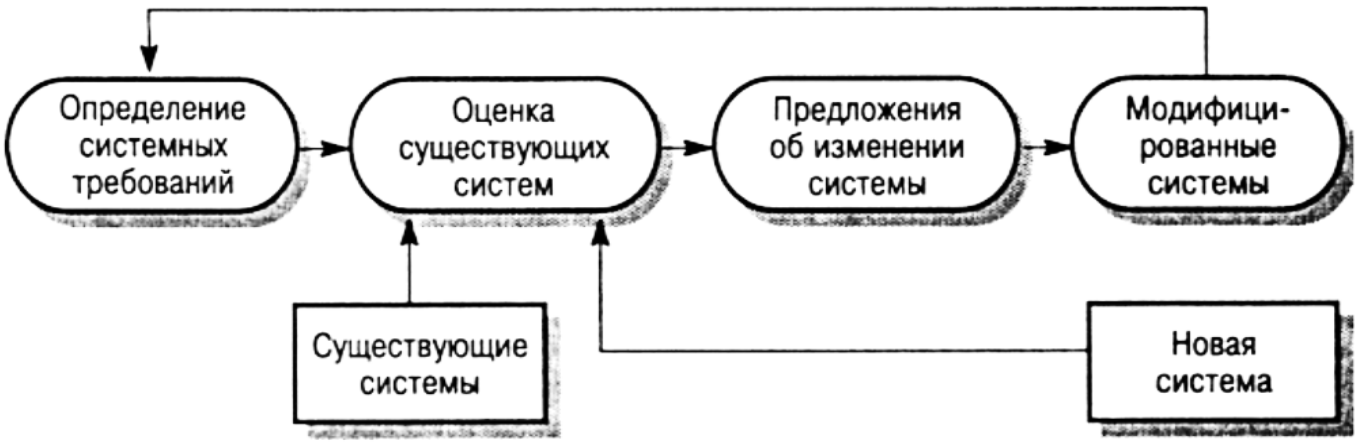
Технология программирования (технология разработки ПО) – это наука об оптимальных способах (приёмах) проведения процесса программирования. Эти способы должны обеспечить создание программного продукта с заданными свойствами в заданных условиях.

8. Эволюция программного продукта. Основные определения, понятия, отличительные черты.

Эволюция ПС – систематическая трансформация существующей системы с целью улучшения ее характеристик качества, поддерживаемой ею функциональности, понижения стоимости ее сопровождения (устранение ошибок), вероятности возникновения значимых для заказчика рисков, уменьшения сроков работ по сопровождению системы.

Причины:

1) Изменения в технической и деловой области.
2) Внесение изменений в подсистемы.
3) Пересмотр ранее принятых решений.
4) Структура системы со временем нарушается из-за сделанных ранее изменений.



Программа – это последовательность команд (или операторов), которая после декодирования её ВМ и транслятором выполняет некоторую работу.

ПО – это группа взаимосвязанных и взаимодействующих модулей или программ.

Программный продукт – это надёжная и эффективная программа (или ПО), которой любой желающий может воспользоваться.

Программное изделие – программный продукт, который удовлетворяет определённому стандарту.

Программная система – программный продукт + что-то ещё (например, соответствующая платформа, ОС, сервисное ПО и др.)

Пакет прикладных программ (ППП) – программный комплекс, который предназначен для автоматизации решения определённого класса задач. ППП должен покрывать конкретную предметную область (т.е. для любой задачи в данной области может быть построена программа (или совокупность программ), которая решает эту задачу).

9. Понятие «модуль» в программировании. Различные виды модулей при использовании основных методов проектирования ПС.

Модульный принцип является общепринятым в практике программирования, т.е. это обязательное условие любой серьёзной программной разработки.

Разбивать ПО на модули необходимо для упрощения процесса разработки и для уменьшения его сложности. Это происходит за счет следующих факторов:

1) Возможности организации коллективной разработки ПС.
2) Использование созданных ранее модулей в новых разработках (появились библиотеки модулей).

Что можно разбивать на модули:

a. Тексты программ на языках программирования.
b. Тексты указаний, применяемых к ПО
c. Тексты исходных данных
d. Сопроводительную документацию
e. Проект системы (диаграммы, схемы, рисунки, тексты и т.д.)

Модуль – это выделенная (по тем или иным причинам) часть первичного программного фонда (Майерс). При использовании различных методов проектирования (и ЯП, соответственно) модулем могут называться различные части исходного текста.

При использовании ***традиционных методологий*** и методов SD и DD:

Модуль – это относительно независимая часть программы (процедура, функция, подпрограмма и т.п.), которая имеет один вход и один выход, имеет небольшой размер и выполняет одну-две функции (Фокс).

При использовании ***объектно-ориентированного метода*** разработки ПС:

Модуль – это отдельный файл (Буч). Еще раньше модулем называли файл, который можно отдельно транслировать (OPascal, Modula).

В новых объектно-ориентированных языках (C++, Java и т.д.) в файлах могут быть различные структурные единицы (классы целиком, только определения классов, интерфейсы и реализации интерфейсов в отдельных файлах и т.п.).

Поэтому в ***стандарте UML*** введено новое понятие – компонента. По стандарту ***компонента*** – это часть модели ПС на физическом уровне (ещё называют физическим модулем). При этом выделяют 2 типа компонент: исполняемые и библиотеки исходного кода.

10. CASE – технологии (инструменты, системы, средства). Эволюция CASE – средств, классификация, характеристики современных CASE – инструментов. Перспективы развития.

CASE-технологии (Computer Aided Software Engineering) – это совокупность методических материалов, автоматизированных методов и инструментальных средств, применяемых для разработки ПО как пригодного к коммерческому распространению продукта. Общая цель всех CASE-средств – поддержка любых процессов (на всех этапах) создания ПО.

Эволюция CASE-средств. Сегодня характерно использование CASE-средств двух поколений:

CASE-I (1984-1990): Главная цель – ***отделение процесса проектирования ПО от его кодирования***. Поэтому развитие этого поколения CASE шло в следующих направлениях:

1. структурное проектирования ПО;
2. разработка средств моделирования данных и структур баз данных;
3. создание средств проектирования спецификаций ПО.

CASE-II (1991—): направлены на ***автоматизацию процессов всего жизненного цикла*** ПО. Компоненты CASE-II могут быть разделены на несколько функциональных групп, решающих определенные задачи (планирование проекта, изучение возможностей решения проблемы, определение требований, системное проектирование, программирование, тестирование и отладка ПО, измерение качества, поддержка документирования, управление процессом проектирования, сопровождение ПО).

Классификация CASE-средств. В настоящее время существует более 500 CASE-средств. В практике используются две классификации:

- ***по функциональному ориентированию***: стратегическое планирование прикладных систем; управление в поддержка полного ЖЦ ПО; управление качеством ПО; управление проектированием; анализ и проектирование ПО; генерация кода; тестирование и отладка; словари данных; создание прототипов; объектно-ориентированная обработка; повторная разработка ПО.
- ***по функциональной наполненности***: категории (tools, toolkit, workbench), типы (группирует средства, применяемые в одних и тех же или близких стадиях) и виды (верхний – модель предметной области, средний – анализ и проектирование, нижний – генерация программ, документация, тестирования).

Характеристики современных CASE-инструментов:

- поддержка полного ЖЦ ПО и полнота интеграции;
- возможность создания прототипов;
- наличие центральной базы данных проекта (repository);
- использование стандартных средств и методов;
- открытость архитектуры CASE;
- простота использования в сочетания с мощными функциями.

Перспективы развития CASE-средств:

- использование баз знаний и искусственного интеллекта;
- интеграция CASE на поддержку полного ЖЦ с автоматическим получением машинных программ по исходным требованиям;
- стандартизация CASE.

11. Роль CASE – инструментов в объектно-ориентированной методологии разработки ПС. Связь CASE – технологии с методами быстрой разработки приложений (RAD).

CASE-технология позволяет отделить проектирование ПО от программирования и отладки. Разработчик с помощью CASE-инструментов занимается проектированием на более высоком уровне, не отвлекаясь на мелкие детали.

Основными понятиями OOD являются ключевые абстракции и механизмы, поэтому необходимы инструменты, содержащие соответствующий набор условных обозначений.

Типы инструментальных средств, полезных при OOD (Буч):

1) Графические автоматизированные системы со встроенными элементами нотаций. Используются на ранних этапах процесса разработки ПО (анализ требований и проектирование). Многие позволяют осуществлять "обратную связь", т.е. восстанавливать по исходному тексту структуру классов и модулей проектов.
2) Инструменты просмотра структуры классов и модулей архитектуры системы (разработчику может понадобиться найти определение класса, которому принадлежит тот или иной объект). Есть почти во всех ОО-языках.
3) Пошаговые трансляторы (обеспечивают трансляцию отдельных операторов и функций). Необходимы для отладки.
4) Инструменты контроля за версиями проекта (обычно единица контроля – модуль).
5) Библиотекари классов (позволяют сортировать классы и модули по определенному принципу/признаку, добавлять классы в библиотеку и находить нужные для повторного использования).

CASE-технология связана с методами визуального программирования или методами быстрой разработки приложений ***RAD*** (Rapid Application Development). Общие черты: применение графических интерфейсов для построения программ, объектных методов, современных способов тестирования (верификации). На смену CASE пришли RAD-средства. В связи с этим рынок CASE-продуктов предлагает облегченные инструменты, которые имеют, прежде всего, средства моделирования данных. Пользователи сами пишут значительную долю программного кода. Другой подход предусматривает использование репозиториев, поддерживающих различные пакеты RAD. Такой подход позволяет найти золотую середину между старой и новой методологией и удачно их сочетать.

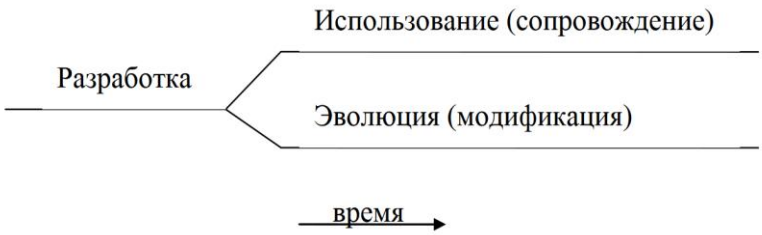
12. Жизненный цикл ПО (ЖЦ). Структура ЖЦ, основные фазы ЖЦ.

ЖЦ ПО – это непрерывный процесс, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его изъятия из эксплуатации.

Стандарт ISO/IEC 12207 определяет структуру ЖЦ, его фазы, действия и задачи, которые должны быть выполнены в процессе разработки и эволюции ПО.

Структура ЖЦ ПО по стандарту базируется на трех группах процессов:

1. Основные процессы (фазы):
   a. разработка;
   b. использование (сопровождение – устранение ошибок);
   c. эволюция (модификация – внесение изменений в систему при изменении требований).

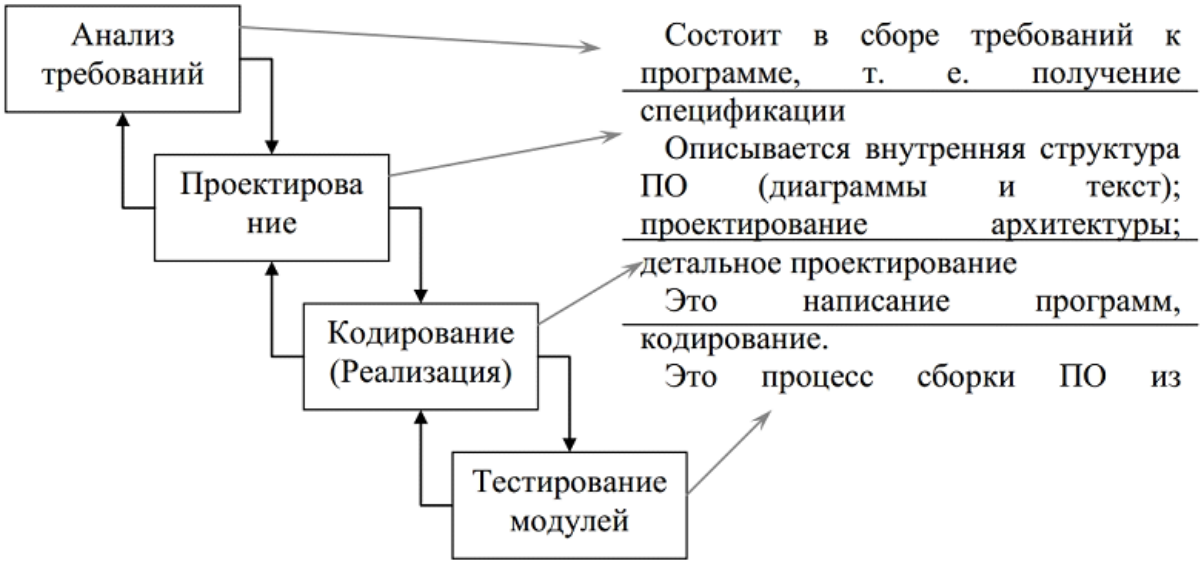


2. Вспомогательные процессы (направлены на поддержку основных процессов, особенно при внесении изменений)
   a. документирование;
   b. обеспечение качества ПО;
   c. верификация;
   d. аттестация;
   e. оценка;
   f. управление конфигурацией.
3. Организационные процессы (связаны с планированием работ, созданием коллектива, контролем за сроком и качеством выполнения, выбором CASE-инструментов)
   a. управление проектом;
   b. улучшение самого ЖЦ;
   c. обучение и другие.

13. Классические модели процесса разработки ПС (каскадная, спиральная)

До 60 г. использовали “двухфазную” модель разработки: кодирование и отладка. До 70 г. “пошаговую”: разработка → проектирование → кодирование → тестирование → внедрение.

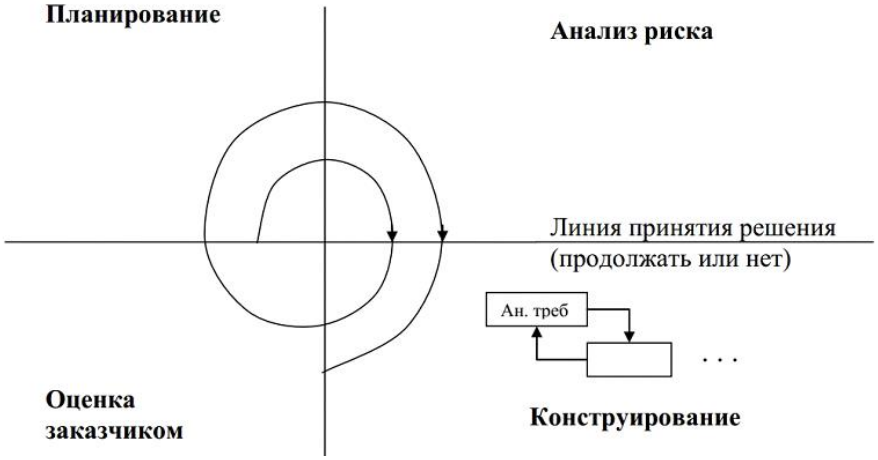
“Каскадная” модель (модель “водопада”): предложена в 1970 г. У. Ройсом. Схематично:



Преимущества относительно пошаговой:

1. Фазы изображены в виде лесенки, т. е. фазы частично перекрываются и любую из фаз можно начинать до того, как будет полностью завершена предыдущая.
2. Появились петли обратной связи между этапами, т.е. есть возможность вернуться на этап выше, если необходимо.
3. Ройсом предлагается параллельно с анализом требований и проектированием разработать прототип (на практике (и на схеме) не обязательно).
4. Возросла роль анализа требований.
5. Каждый этап завершается выпуском документации, достаточной для того, чтобы продолжить разработку на следующем этапе, причем вся документация должна быть согласована и утверждена.

Спиральная модель. Предложена Боэмом в 1988 г. Базируется на лучших свойствах классической модели и макетирования, к которым добавляется новый элемент – анализ риска и новый этап – системный анализ. Схематично:



1. Планирование – определение целей, требований, ограничений, составление плана разработки (начиная со 2-го витка планирование проводится на основе оценки заказчика).

2. Анализ риска (на 1-м витке – на основе начальных требований, на следующих – на основе реакции заказчика).

3. Конструирование – разработка ПС (на 1-м витке – начальный макет, на 2-м – следующий уровень макета, на последнем – готовая система).

4. Оценивание – оценка заказчиком текущих результатов конструирования.

Получается, что с каждым витком спирали строятся все более полные версии ПО (и по функциональности, и по эффективности).

Достоинства спиральной модели:

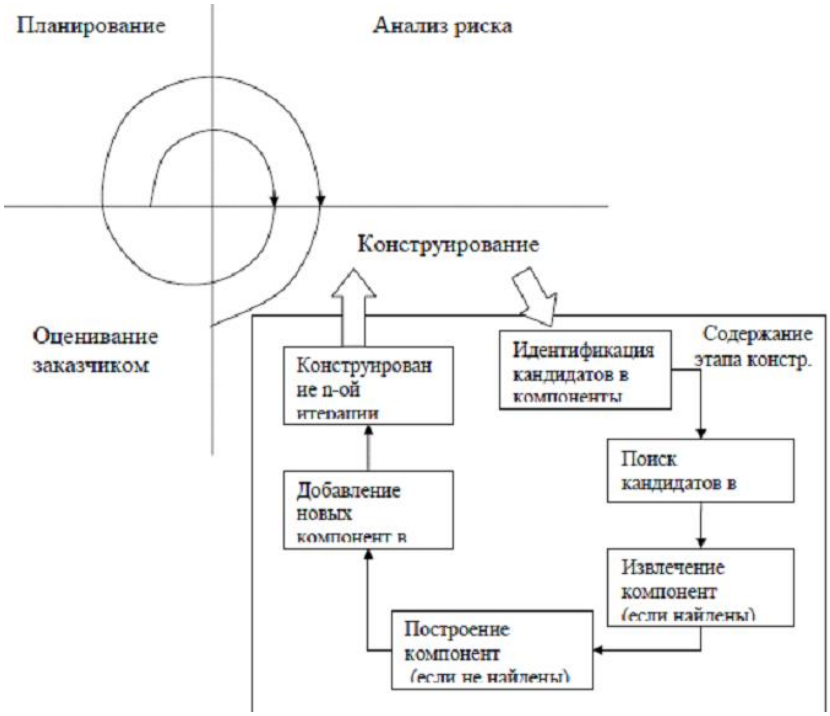
- наиболее реально (в виде эволюции) отображает разработку ПО;
- позволяет явно учитывать риск на каждом витке эволюции разработки;
- использует моделирование для уменьшения риска.

Недостатки спиральной модели:

- повышенные требования к заказчику;
- трудности контроля и управления временем разработки.

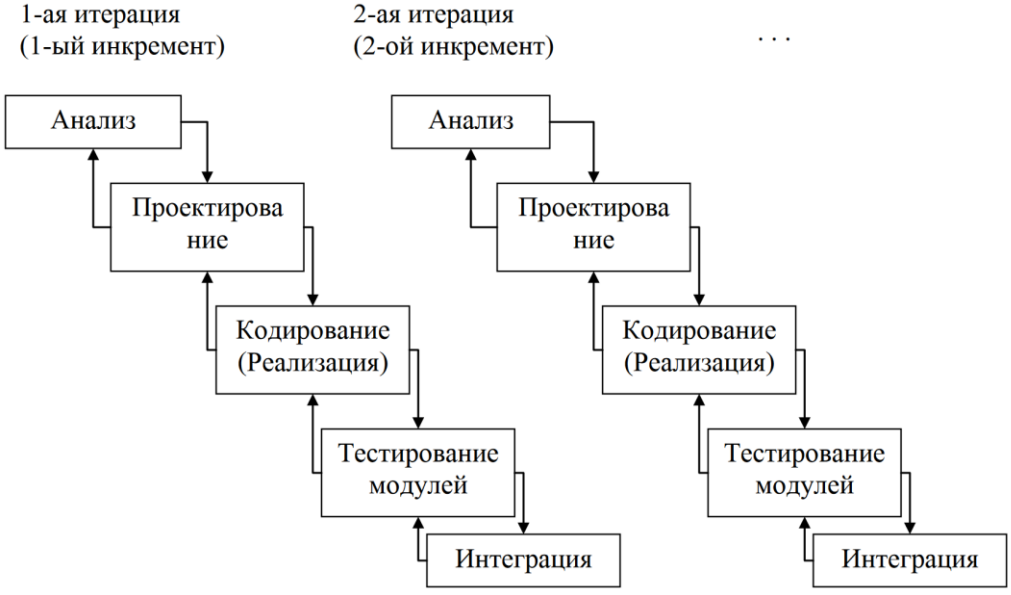
14. Модели процесса разработки ПС (компонентно-ориентированная, инкрементная, RAD-модель).

Компонентно-ориентированная модель является развитием спиральной модели. В этой модели конкретизируется содержание квадранта конструирования. Внимание уделяется повторному использованию существующих программных компонентов (библиотечных). ***Достоинства***: уменьшает время и стоимость, увеличивает производительность разработки ПС.



Инкрементная модель. Предложена Кратченом в 1995 г., усовершенствована Ройсом.

В начале процесса конструирования определяются почти все пользовательские и системные требования. Затем конструирование выполняется в виде последовательности итераций. Каждая итерация – “мини-водопад”. На первой итерации реализуется самые важные или самые рискованные запланированные функции. Следующая версия реализует дополнительные возможности и т. д., пока не будет получена полная система (реализует все запланированные функции). На каждой итерации получается работающий продукт, который реализует часть функциональных требований.



RAD (Rapid Application Development) – это ЖЦ процесса проектирования, созданный для достижения более высокой скорости разработки и качества ПО. RAD предполагает, что разработка ПО осуществляется небольшой командой разработчиков за срок порядка трех-четырех месяцев путем использования инкрементного прототипирования с применением инструментальных средств визуального моделирования и разработки. Технология RAD предусматривает активное привлечение заказчика уже на ранних стадиях.

Преимущества: высокая скорость разработки, низкая стоимость, высокое качество.

Важные моменты:

1. Инструментарий должен быть нацелен на минимизацию времени разработки.
2. Создание прототипа для уточнения требований заказчика.
3. Цикличность разработки: каждая новая версия продукта основывается на оценке результата работы предыдущей версии заказчиком.
4. Минимизация времени разработки версии осуществляется за счёт переноса уже готовых модулей и добавления функциональности в новую версию.
5. Команда разработчиков должна тесно сотрудничать, каждый участник должен быть готов выполнять несколько обязанностей.
6. Управление проектом должно минимизировать длительность цикла разработки.

15. Тяжеловесные и облегченные (гибкие) модели процессов разработки ПС.

Все современные методологии условно можно разделить на 3 категории:

1. тяжелые (прогнозируемые);
2. легкие (гибкие, подвижные);
3. средние (промежуточные);

Упрощенно, каждая из них предназначена для работы в условиях больших, малых и средних проектов.

Основная характеристика ***тяжеловесной методологии*** (UP, RUP2) – весь объем предстоящих работ планируется и организуется для обеспечения прогнозируемости процесса. Тяжеловесная методология применима при более-менее фиксированных требованиях и многочисленной группе разработчиков разной квалификации.

Облегченные методологии (SCRUM, XP) используются при:

1. Частых изменениях требований;
2. Малочисленной группе высококвалифицированных разработчиков;
3. Грамотном заказчике, который согласен участвовать в процессе разработки.

Средние методология (ICONIX) представляет собой нечто среднее между тяжелыми и гибкими методологиями. Основная характеристика: масштабируемость, т.е. процесс разработки может быть настроен на работу как в малой команде над небольшим проектом, так и в большой команде над большим и серьезным проектом.

При ***выборе методологии*** надо определить, что надо оптимизировать:

1. Процесс управления/разработки → облегченные.
2. Процесс разработки, то ближе к легким.
3. Больше команда → тяжелее методология

Можно выделить основные типы современных проектов:

1. Проекты для постоянных заказчиков (в основном agile)
2. Продукты под заказ (средние или ближе к тяжелым)
3. Тиражируемый продукт (средние или ближе к тяжелым)
4. Аутсорсинг (предпосылки к утяжелению проекта)

16. Унифицированный процесс разработки ПС. Модель процесса RUP.

Унифицированный процесс – это обобщенный каркас процесса разработки, который может быть специализирован для широкого круга ПС, различных областей применения, уровней компетенции и размеров проекта. Основные характеристики:

1. управляется вариантами использования. ***Вариант использования*** – это часть функциональности системы, необходимая для получения пользователем значимого для него результата. Все варианты использования в совокупности составляют модель вариантов использования, которая описывает полную функциональность системы.
2. архитектурно-ориентированный. Архитектор в процессе работы:
   a. Создает грубый набросок архитектуры.
   b. Работает с подмножеством выделенных вариантов использования.
   c. Созданная архитектура – это база для разработки других вариантов использования.
3. итеративный (разделение работы на небольшие части) и инкрементный: снижает риски, ускоряет разработку, адаптирована к изменениям.

Жизненный цикл UP разбивается на циклы, каждый из которых завершается поставкой продукта. Каждый цикл развития состоит из четырех фаз – анализа требований, проектирования, построения и внедрения. Каждая фаза подразделяется на итерации.

Основные элементы UP:

1) ***Исполнитель*** (worker) – это роль или роли, определяющие поведение и обязанности лица или группы;
2) ***Деятельность*** – это часть работы в рамках процесса разработки программного обеспечения, которую выполняет некоторый исполнитель. Виды деятельности могут быть разбиты на этапы, которые можно объединить в основные группы:
   a) Этапы обследования − исполнитель исследует исходные артефакты, определяет результирующие артефакты.
   b) Этапы производства − исполнитель создает или изменяет некоторые артефакты.
   c) Этапы рецензирования − исполнитель проверяет результаты.
3) ***Артефакт*** – это общее название для любых существенных видов информации: порождаемой, модифицируемой или используемой процессом. Можно выделить два основных типа артефактов − ***технические артефакты*** (создаваемые в ходе различных фаз процесса) и ***артефакты управления*** (бизнес-план, план разработки и подбора исполнителей).
4) ***Актант*** (actor) − некто (или нечто) вне системы, взаимодействующее с системой.
5) ***Модель*** – абстракция, описывающая систему с определенной точки зрения и на определенном уровне абстрагирования.
6) ***Вариант использования*** (use case, прецедент) − определение набора последовательностей действий, включая варианты, при которых система приносит отдельному актанту полезный и понятный результат.
7) ***Рабочий процесс*** – набор видов деятельности и связанные с ними наборы исполнителей и артефактов.

RUP (Rational Unified Process) – немного улучшенная компанией Rational версия, ядром остается UP. Упор делается на архитектурные риски.

17. XP – процесс (экстремальное программирование).

Экстремальное программирование (Кент Бэк) – облегченная (гибкая) методология. XP-процесс ориентирован на ***группы малого и среднего размера***, строящие программное обеспечение ***в условиях неопределенных или быстро изменяющихся требований***.

Основная идея XP – устранить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов и реляционных БД. XP-группа имеет дело с изменениями требований на всем протяжении итерационного цикла разработки, причем цикл состоит из ***очень коротких итераций***.

Базовые действия в XP-цикле:

- кодирование,
- тестирование,
- выслушивание заказчика,
- проектирование.

Динамизм обеспечивается с помощью четырех характеристик:

1. непрерывной связи с заказчиком,
2. простоты,
3. быстрой обратной связи (с помощью модульного и функционального тестирования),
4. смелости в проведении профилактики возможных проблем.

Сегодня в XP популярны **2 лозунга**:

- “Ищите самое простое решение, которое может сработать”;
- “Это вам не понадобится”.

Кент Бек приводит **4 критерия простой системы**:

1) Система успешно проходит все тесты;
2) Код системы явно раскрывает все изначальные замыслы;
3) В программе отсутствует дублирование кода;
4) Используется минимально возможное количество классов и методов.

Ценности XP: уважение, общение, простота, смелость, обратная связь.

Принципы XP: человечность, взаимная выгода, сходство, постоянное улучшение, разнообразие, обдумывание, поток, возможности, избыточность, неудачи, качество, маленькие шаги, ответственность.

Практики XP относящиеся к анализу требований и планированию (а), человеческому фактору (б), проектированию (в), программированию и выпуску продукта (г):

а) ***истории пользователя***, ***недельный ритм***, ***квартальный ритм***, ***принятие своей слабости***; *непосредственное вовлечение заказчика*, *инкрементная поставка продукта (всё потихоньку)*, *контракт с оговоренным объёмом работ*;

б) ***работа в одном помещении***, ***информативное окружение***, э***нергичная работа без овертаймов***, ***парное программирование***; *постоянство (например, команды)*, *“усушка и утряска” (?)*;

в) ***инкрементальное проектирование***, ***тесты вперёд***; *анализ причин*;

г) ***десятиминутная сборка***, ***постоянная интеграция***; *код и тесты*, *коллективное владение кодом*, *единая база программного кода*, *ежедневная поставка системы*.

18. SCRUM – модель процесса разработки.

В последнее время при разработке проектов все большую популярность набирают различные agile-технологии, в основе которых лежит итеративный процесс разработки с функционирующей версией продукта после каждой итерации.

Термин Scrum был впервые озвучен в работе Х. Такеучи и И. Нонака.

Scrum – одна из самых популярных методологий гибкой разработки. ***Scrum определяет***:

1) правила, по которым должен планироваться и управляться список требований к продукту;
2) правила планирования итераций;
3) основные правила взаимодействия участников команды;
4) правила анализа и корректировки процесса разработки.

Итерация: планируем → фиксируем → реализуем → анализируем.

Концепция Scrum: три роли, три практики и три основных документа.

Роли

Product Owner. Отвечает за разработку продукта (представитель заказчика). Составляет бизнес-план и план развития с требованиями, отсортированными по значимости, принимает окончательные решения. ***Обязанности***:

- Отвечает за формирование product vision;
- Управляет ожиданиями заказчиков;
- Координирует и приоритезирует Product Backlog;
- Предоставляет понятные и тестируемые требования к команде;
- Взаимодействует с командой и заказчиком;
- Отвечает за приемку кода в конце каждой итерации.

Scrum Master. Должен быть одним из членов команды разработки и участвовать в проекте как разработчик. ***Обязанности***:

- обеспечение максимальной работоспособности и продуктивности команды, четкого взаимодействия между всеми участниками проекта;
- своевременное решение всех проблем, тормозящих или останавливающих работу любого члена команды;
- ограждение команды от всех воздействий извне во время итерации и обеспечение следования процессу всех участников проекта;
- делает проблемы и открытые вопросы видимыми;
- отвечает за соблюдение практик и процесса в команде.

Scrum Team – группа, состоящая из 5–9 самостоятельных, инициативных программистов. Задачи: поставить реально достижимую и значимую цель для итерации и сделать все для того, чтобы эта цель была достигнута в отведенные сроки и с заявленным качеством. ***Обязанности***:

- Оценка элементов Backlog;
- Принятие решений по дизайну и имплементации;
- Разработка софта и предоставление его заказчику;
- Отслеживание собственного прогресса;
- Отчитывание перед Product Owner.

Практики

Sprint – итерация длительностью 1 месяц. Результатом Sprint является готовый продукт (build), который можно передавать заказчику. Подготовка к первой итерации, называемой спринт (Sprint), начинается после того, как владелец продукта разработал план проекта, определил требования. Такой список требований называется *журналом продукта (Product Backlog)*. При планировании итерации происходит детальная разработка сессий планирования

спринта (Sprint Planning Meeting). Scrum-команда проверяет оценки требований, убеждается, что они достаточно точны, чтобы начать работать, решает, какой объем работы она может успешно выполнить за спринт, основываясь на размере команды, доступном времени и производительности. Важно, чтобы Scrum-команда выбирала первые по приоритету требования из журнала продукта. После того как Scrum-команда обязуется реализовать выбранные требования, Scrum-мастер начинает планирование спринта. Scrum-команда разбивает выбранные требования на задачи, необходимые для реализации. Эта активность в идеале не должна занимать больше четырех часов, и ее результатом служит список требований, разбитый на задачи, – *журнал спринта (Sprint Backlog)*.

Daily Scrum Meeting. Этот митинг проходит каждое утро в начале дня. Он предназначен для того, чтобы все члены команды знали, кто и чем занимается в проекте. Скрам-митинг проводит Скрам-Мастер. Он по кругу задает вопросы каждомучлену команды: Что сделано вчера? Что будет сделано сегодня? С какими проблемами столкнулся? Скрам-Мастер собирает все открытые для обсуждения вопросы в виде Action Items, например в формате что/кто/когда

Sprint Review Meeting. В конце каждого спринта проводится демонстрационный митинг. Сначала Scrum-команда демонстрирует владельцу продукта сделанную в течение спринта работу. Владелец продукта определяет, какие требования из журнала спринта были выполнены, и обсуждает с командой и заказчиками, как лучше расставить приоритеты в журнале продукта для следующей итерации. Затем цикл замыкается, и начинается планирование следующего спринта.

Sprint Abnormal Termination. Остановка спринта производится в исключительных ситуациях: команда не может достичь цели спринта в отведенное время, необходимость в достижении цели спринта исчезла. После остановки спринта проводится митинг с командой, где обсуждаются причины остановки спринта. После этого начинается новый спринт.

Артефакты

Product Backlog. В начале проекта владелец продукта готовит журнал продукта – список требований, отсортированный по значимости, а Scrum-команда дополняет этот журнал оценками стоимости реализации требований.

Sprint Backlog. Журнал спринта содержит функциональность, выбранную владельцем продукта из журнала продукта. Все функции разбиты по задачам, каждая из которых оценивается командой. Разбивка на задачи поможет так спланировать итерацию,чтобы в конце не осталось ни одной невыполненной задачи и, соответственно, достичь ее цели.

Burndown Chart. График спринта показывает ежедневное изменение общего объема работ, оставшегося до окончания итерации.

Плюсы*:*

1) Простота и легковесность;
2) Использование этой методологии дает возможность выявлять и устранять отклонения от желаемого результата на более ранних этапах разработки программного продукта.
3) Вовлеченность каждого члена команды (выставление оценок, полностью осведомлен о состоянии проекта, участвует в принятии решений);
4) Плотная работа с заказчиком, принимает непосредственное участие;

19. ICONIX – процесс.

ICONIX – средняя методология. Процесс ICONIX, как и UP, основан на прецедентах (вариантах использования). В этом процессе также применяется язык моделирования UML, но используется минимальное подмножество этого языка, которого часто оказывается достаточно для работы над проектом.

В основу процесса ICONIX положены четыре ***основных этапа разработки ПО*** на основе вариантов использования:

1. моделирование предметной области;
2. моделирование прецедентов;
3. анализ пригодности требований (проверка на выполнение всех функциональных требований);
4. построение диаграмм последовательности. Это то, что отличает ICONIX от других моделей процесса разработки: переход от прецедентов («что делать») к диаграммам последовательности («как делать»).

Важный момент: до описания модели прецедентов должен быть создан ***прототип*** графического интерфейса пользователя, который можно показать заказчику и уточнить.

Вся сложность процесса моделирования ПС состоит в том, как перейти от нечетких формулировок прецедентов к очень точным и детальным диаграммам последовательности. Для этого на первом этапе в ICONIX используют так называемые ***диаграммы пригодности***, которые в UML называются кооперативными диаграммами. В диаграммах пригодности используются стереотипы объектов.

Следующим этапом в ICONIX проводится моделирование предметной области. Это своего рода словарь основных абстракций – важнейших сущностей задачи. Такие существительные (описывают основные понятия из предметной области) называются ***доменными объектами***.

20. Прототип (макет) программной системы (виды, достоинства и недостатки макетирования). Масштаб проекта и риски.

Программный прототип (макет) – это ранняя реализация системы, в которой демонстрируют только часть ее функциональных возможностей. Прототипы необходимы для снятия неопределенности в требованиях заказчика. Выбор прототипа зависит от проблемы, которую надо решить. Цель – создать прототип с наименьшими затратами.

Дэвис выделяет следующие ***категории прототипов***:

1. Отбрасываемые (созданы, чтобы опробовать какое-либо архитектурное решение: осуществимо оно или нет).
2. Эволюционирующие (прототип развивается с той же архитектурой, которая будет использоваться в конечной версии продукта).
3. Операционные.
4. Вертикальные (срез функциональности приложения от интерфейса пользователя до сервисных функций).
5. Горизонтальные (реализуются не все слои архитектуры, но воплощаются особенности интерфейса пользователя).
6. Интерфейсные (много решений, чтобы строить в программах интерфейсы, уже не очень актуально).
7. Алгоритмические.

Достоинства прототипов:

1. Обеспечивают определение более полных требований к ПО.
2. Обеспечивают снятие неопределенностей.

Недостатки прототипов:

1. Заказчик может принять макет за продукт. «Немножко подправьте и все».
2. Разработчик может принять макет за продукт. «И так сойдет».

Масштаб проекта и риски. Задать “масштаб” проекта – оценить ресурсы проекта. Масштаб определяется следующими тремя переменными:

1. Набором функций, которые необходимы пользователю.
2. Ресурсами (труд команды разработчиков), которыми располагает проект.
3. Временем, которое выделено на реализацию.

Масштаб проекта часто задают в виде “прямоугольника”.

Если объем работ, необходимых для реализации функций системы, равен ресурсы∗время, то говорят, что проект имеет достижимый масштаб. ***«Закон Брукса»***: добавление ресурсов в проект почти всегда приводит к увеличению времени (масштаба проекта).

Риск – вероятность того, что реализация функции окажет негативное влияние на график и бюджет. Своевременное выявление риска и принятие соответствующих мер позволяют предотвратить срыв проекта. Типы рисков: устранимые и неустранимые.

21. Содержание основных рабочих процессов по созданию ПО (анализ требований, системный анализ, проектирование, реализация, тестирование).

Анализ требований – это процесс сбора требований к ПО, их систематизации, документирования, анализа, выявления противоречий, неполноты, разрешения конфликтов в процессе разработки ПО. Полнота и качество анализа требований играют ключевую роль в успехе проекта. Требования к ПО должны быть документируемые, выполнимые, тестируемые, с уровнем детализации достаточным для проектирования системы. Требования могут быть функциональными (*что* необходимо реализовать в продукте) и нефункциональными (*как* должна работать система и какими свойствами она должна обладать).

Системный анализ («что») – последовательность действий по установлению структурных связей между переменными или элементами проектируемой системы.

Проектирование ПО («как») – процесс создания проекта ПО. Проектирование подразумевает выработку свойств системы на основе анализа постановки задачи, а именно: моделей предметной области, требований к ПО. Проектированию обычно подлежат архитектура ПО, устройство компонентов ПО, пользовательские интерфейсы.

Реализация заключается в создании ПО в соответствии с архитектурой системы и технологией создания системы, определенными на этапах обследования и технического проектирования.

Тестирование ПО – процесс исследования ПО с целью получения информации о качестве продукта. Качество можно определить как совокупную характеристику исследуемого ПО с учётом следующих составляющих: надёжность, сопровождаемость, практичность, эффективность, мобильность, функциональность.

22. Организационные процессы (распределение ресурсов, управление проектом, способы организация коллектива разработчиков). Роли разработчиков в проекте.

Распределение ресурсов. Приблизительные временные затраты на реализацию этапов разработки ПО: 15% – системный анализ, 20% – проектирование, 15% – программирование, 40% – тестирование и сборка, 10% – документирование.

Использование объектно-ориентированного метода разработки позволяет уменьшить количество требуемых ресурсов и изменить временное соотношений между различными этапами (больше на анализ и проектирование). Не учитывается документирование, так как при ООП документация разрабатывается в процессе.

Управление проектом. Главной задачей руководства является поддержание целостности основной идеи в ходе работ, а также контроль за качеством продукта. Для контроля качества используется метод сквозного (структурного) контроля. Данный метод представляет собой серию проверок, проводимых на разных этапах цикла разработки, и состоит в регулярных встречах разработчиков ПО. Он позволяет обнаруживать и исправлять ошибки как можно раньше. Контроль осуществляется посредством специальных контрольных сессий. Обнаруженные ошибки соответствующим образом документируются (контрольный лист).

Способы организации коллектива разработчиков:

Ранее использовались методы “монгольской орды” (сначала малая группа, потом по необходимости подключаются остальные) и “суперпрограммиста”. Не актуальны из-за усложнения разработки.

1. ***Метод “бригады главного программиста”***: главному программисту дается группа специалистов (6-7 человек). Ядро бригады составляют 3 человека: главный программист, его помощник и библиотекарь. Главный программист – это опытный специалист высокой квалификации, полностью отвечает за разработку проекта. Помощник главного программиста – тоже квалифицированный специалист, при необходимости может заменить главного программиста. Библиотекарь осуществляет корректировку программ и их прогон, ведет библиотеку листингов, исходных и объектных модулей.
2. ***Метод “хирургической бригады”***. Главный программист играет роль, аналогичную роли оперирующего хирурга, опираясь на группу специалистов, члены которой скорее ассистируют ему, чем независимо пишут составные части ПО.
3. ***“Демократическая бригада”***. Не имеет формального лидера, обязанности в бригаде распределяются начальством (менеджером) в соответствии со способностями и опытом ее членов. Коллектив такой бригады (в отличие от первых двух методов) сохраняет свой состав, переходя от проекта к проекту.

Роли разработчиков. При объектно-ориентированном подходе (Буч):

1. Архитекторы систем.
2. Проектировщики классов.
3. Специалисты, реализующие внутреннее строение классов.
4. Программисты-прикладники.

23. Документирование программного продукта. Различные виды документов, их содержание. Виды документов при использовании объектно-ориентированной методологии разработки ПС.

Документация – это самое важное из того, что должны сделать разработчики ПО. Документация может содержать:

- хорошо прокомментированный текст программы;
- всевозможные схемы (диаграммы), полученные на этапе проектирования ПО.

Продукты традиционных методов проектирования (SD, DD) – это диаграммы разбиения ПО на модули, схемы модулей, диаграммы процессов. В документацию традиционных методов проектирования (SD, DD) входят описание данных для всей системы (подсистем) и описание входных и выходных данных для каждого модуля.

Продукты OOD – это диаграммы классов, модулей, процессов и объектные диаграммы. В документации они должны быть сгруппированы определенным образом. Проект каждого значительного сегмента системы (как правило, подсистемы) должен быть описан в отдельном документе, который отражает логическую схему системы и имеет следующую структуру:

1. Требуемые функции;
2. Диаграммы классов и объектов;
3. Базовые элементы классов и объектов;
4. Диаграммы модулей и объектов;
5. Базовые элементы модулей и процессов;
6. Результаты в виде работающих прототипов.

Большинство разделов может быть написано с помощью полуавтоматических методов.

Виды документации (могут быть и другие):

- Документация, имеющая первые три раздела, обычно носит название *“Руководство программиста”*.
- В *“Руководстве пользователя”* (для больших систем – *“Руководстве оператора”*) необходимо описать, что, когда и при каких обстоятельствах делать. Пользователь должен иметь достаточно информации о том, что, как и почему происходит в системе.
- Документация для менеджеров пользователей называется *“Общее описани*е” (*“Концепции и возможности”)*. В такой документации описывается, что система умеет делать.
- В период эксплуатации системы может возникнуть необходимость настройки ее на новую платформу (новый тип ЭВМ, ОС). Для таких случаев разработчик пишет специальный документ *“Руководство системного программиста”*.

Исходя из практики, в комплекс необходимой документации обычно входят:

1. *Техническое описание* (назначение, технические характеристики, принципы построения).
2. *Справочное руководство* (управление, команды, сообщения об ошибках и т.д.)
3. *Рекламный буклет* (краткое описание, отличия от аналогов и т.д.)
4. *Руководство пользователя*

Для некоторых ПП бывает достаточно только “Руководства пользователя”.

Важной характеристикой программного продукта является система обучения пользователей (групповое, индивидуальное и автоматизированное).

24. Методы и средства структурного анализа.

Структурный анализ – это метод исследования системы, при котором анализ начинается с общего обзора системы, а затем идет детализация по уровням (иерархическое дерево).

Для таких методов ***характерно***:

1. разбиение на уровни абстракции с ограничением числа элементов на каждом из уровней (обычно от 3 до 6-7);
2. может присутствовать ограниченный контекст, который включает только существенные на каждом уровне детали;
3. использование строгих формальных правил записи.

Базовые принципы:

- принцип “разделяй и властвуй”;
- принцип иерархического упорядочивания.

Принципы разработки ПО:

1. ***абстрагирования*** (выделение существенных аспектов системы на определенном уровне и отвлечение от несущественных).
2. ***сокрытия*** (каждая часть “знает” только необходимую ей информацию).
3. ***концептуальной общности*** – заключается в следовании единой философии на всех этапах жизненного цикла.
4. ***полноты*** – заключается в контроле отсутствия лишних элементов.
5. ***непротиворечивости*** – заключается в обоснованности и согласованности элементов.
6. ***логической независимости*** – заключается в концентрации внимания на логическом проектировании и независимости от физического проектирования.
7. ***независимости данных*** (модели данных должны быть проанализированы и спроектированы независимо от логической обработки и их физической структуры).
8. ***структурирования данных*** (данные должны быть структурированы и иерархически организованы).

Методы и средства структурного анализа.

Для структурного анализа используются ***три группы средств***, которые иллюстрируют:

1. функции, которые система должна выполнять;
2. отношения между данными;
3. поведение системы, которое зависит от времени.

Таких средств очень много. В методах структурного анализа чаще всего применяются:

1. DFD (Data Flow Diagrams) – диаграммы потоков данных совместно со словарями данных.
2. Спецификации процессов (или миниспецификации).
3. ERD (Entity-Relation Diagrams) – диаграммы “сущность-связь”.
4. STD (State Transition Diagrams) – диаграммы переходов и состояний.

25. Различные подходы проведения анализа объектно-ориентированных систем (классический, на основе поведения, анализ вариантов, анализ предметной области, неформальный, структурный). CRC – карточки.

Классические подходы. Основное внимание уделяют осязаемым элементам предметной области и предлагают начинать анализ системы, составив словарь предметной области, а затем исходя из словаря выделять классы и объекты.

1. ***Подход Шлеера и Меллора***. Кандидаты в классы и объекты: осязаемые предметы, роли, взаимодействие.
2. ***Подход Росса***. Похож на предыдущий, но моделирует ОО базы данных. Кандидаты в классы и объекты: люди, места, предметы, организации, концепции, события.
3. ***Подход Йордана***. Кандидаты в классы и объекты: структуры (общее-целое, часть-целое), устройства, события, роли, места, внешние системы

Анализ поведения (Вирфс-Брок). Анализ системы начинается с анализа ее поведения. Вводится понятие ответственности объекта – это совокупность всех услуг, которые объект может выполнить – контракты. Объекты со схожими ответственностями объединяются в один класс. Затем строится иерархия классов, в которой каждый подкласс должен выполнять обязательства родительского класса, но может иметь и свои услуги. После построения иерархий классов остается выявить классы-клиенты и классы-поставщики услуг и установить между ними соответствующие отношения (ассоциация, зависимость).

CRC-карточки (Class, Responsibility, Collaboration – Класс-Ответственность-Кооперация) – простой и эффективный способ анализа поведения будущей программной системы. Использование карточек позволяет минимизировать сложность дизайна. CRC-карты акцентируют внимание на сущности класса и скрывают детали, рассмотрение которых на данном этапе будет непродуктивным. CRC-карты также заставляют дизайнера воздержаться от назначения классу слишком многих обязанностей.

Анализ предметной области. Суть – изучение всех приложений в рамках данной предметной области.

Анализ вариантов (Шлеер, Меллор). Суть подхода – в перечислении основных сценариев работы системы. Затем сценарии прорабатываются и устанавливается:

1) какие объекты участвуют в сценариях

2) какие обязанности каждого объекта

Проработать только основные потоки недостаточно, необходимо учесть исключительные ситуации, а также учесть дополнительные аспекты поведения объектов системы.

Неформальное описание (Шлеер, Меллор). Задача описывается на обычном языке, затем подчеркиваются существительные и глаголы. Существительные – кандидаты на роль классов, а глаголы могут стать именами основных методов классов.

Структурный анализ – это метод исследования системы, при котором анализ начинается с общего обзора системы, а затем идет детализация по уровням (иерархическое дерево).

26. Методы объектно-ориентированного проектирования ПО (модели).

На этапе проектирования ОО-системы необходимо описать **логическую** и **физическую** модели системы. Причем представить в моделях и ***статические***, и ***динамические*** аспекты проектируемой системы. В основе ОО методологии лежит объектная модель. Она имеет 6 главных принципов: *абстрагирование*, *инкапсуляция*, *полиморфизм*, *наследование*, *иерархия* и *модульность*. Объект – это сущность, которая объединяет данные и функции. Объект может только менять свое состояние, может управляться или может становиться в определенное отношение к другим объектам.

Объектно-ориентированное программирование (ООП) – это методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию (обобщение, агрегация, ассоциация и зависимость). В данном определении можно выделить 3 части:

1) ООП использует в качестве базовых элементов объекты, а не алгоритмы;
2) каждый объект является экземпляром определенного класса;
3) классы организованы иерархически.

Иерархическое отношение классов подразумевает наследование – отношение обобщения в UML. Классы связанных между собой объектов связаны отношением ассоциации. Итак, ОО методология это: OOAnalysis + OODesign + OOProgramming.

На результатах анализа строятся логические и физические модели. Причем эти модели должны показывать не только статику, но и динамику объектов: это диаграммы классов, объектов, состояний, деятельностей, компонентов, процессов. Модели проекта, в свою очередь, создают фундамент для реализации системы на объектно-ориентированном языке.

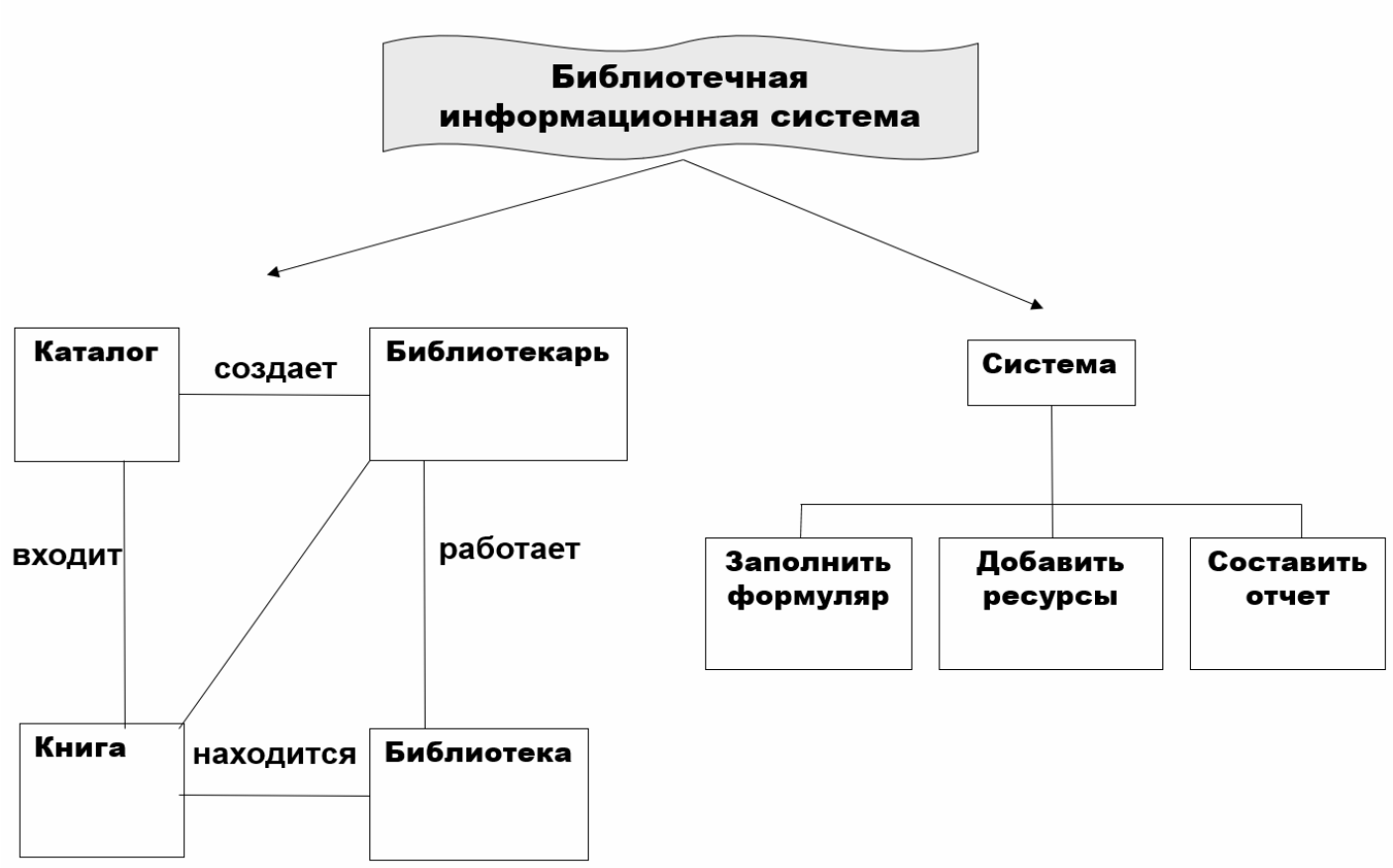
27. Роль декомпозиции в проектировании (алгоритмическая и объектно-ориентированная). Пример библиотечной системы.

При проектировании сложной ПС необходимо разделять ее меньшие подсистемы, каждую из которых можно совершенствовать независимо. Декомпозиция вызвана сложностью программирования ПС.

Алгоритмическая декомпозиция. Разделение алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса.

Объектно-ориентированная декомпозиция. Основана на объектах, а не на алгоритмах.

Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия. Начать разделение системы надо либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения. Объектная декомпозиция уменьшает размер ПС за счет повторного использования общих механизмов.



28. Основные принципы объектной модели (абстрагирование, инкапсуляция, иерархия, модульность, наследование, полиморфизм).

Абстрагирование. Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов. Абстрагирование позволяет отделить самые существенные особенности поведения от несущественных.

Инкапсуляция. Инкапсуляция занимается внутренним устройством объекта. Чаще всего инкапсуляция выполняется посредством скрытия информации и внутренней структуры объекта, реализации его методов. Практически это означает наличие двух частей в классе: ***интерфейса*** и ***реализации***. Так, инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение.

Иерархия. Иерархия – это упорядочение абстракций, расположение их по уровням. Основными видами иерархических структур применительно к сложным системам являются ***структура классов*** (иерархия "is-a") и ***структура объектов*** (иерархия "part of").

Модульность – это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Наследование – механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства к методам и свойствам родительского.

Полиморфизм – возможность объектов с одинаковой спецификацией иметь различную реализацию.

29. Унифицированный язык моделирования ПС (UML). Словарь, достоинства и возможности.

UML вобрал в себя черты нотаций Буча, Румбаха и Якобсона (три друга). Создатели UML представляют его как **язык для написания моделей** анализа, проектирования, реализации, документирования ОО ПС, а также бизнес-систем и любых информационных систем.

Словарь UML образуют 3 разновидности строительных блоков:

1) ***Предметы*** – это абстракции, которые являются основными элементами в модели (класс, объект, интерфейс, кооперация, актер, прецедент, компонент, узел, взаимодействие (сообщения связи), состояние, пакет, комментарии)

2) ***Отношения*** связывают предметы: *зависимость* (изменения в одном влекут изменения в другом), *ассоциация* (описывает набор связей между объектами), *агрегация* (целое и его части), *обобщение* (отношение специализации), *реализация* (между интерфейсом и реализацией).

3) ***Диаграммы*** – графическое представление множества элементов (изображается чаще всего как связный граф из вершин-предметов и дуг-отношений). Теоретически диаграмма может содержать любую комбинацию предметов и отношений, но на практике ограничиваются только девятью видами диаграмм:

1. диаграммы прецедентов (вариантов использования);
2. диаграммы классов;
3. диаграммы объектов;
4. диаграммы последовательностей;
5. диаграммы сотрудничества (кооперации);
6. диаграммы состояний:
7. диаграммы деятельности;
8. диаграммы компонент;
9. диаграммы размещения.

Наличие стандартного языка очень важно для реализации одного из преимуществ визуального моделирования – коммуникации. Именно общение между пользователями и командой разработчиков является основной целью визуального моделирования.

С помощью языка UML можно определить, является ли достигнутое при анализе понимание системы адекватным.

Язык UML предоставляет разработчику средства как для демонстрации своих проектов, так и для чтения и понимания чужих.

30. Механизмы расширения в UML 2.

UML создавался изначально как открытый язык, который допускает расширения. Механизмами расширения в UML являются: ограничения, теговые величины и стереотипы; они позволяют адаптировать UML под новые технологии, под нужды конкретных проектов.

1) **Ограничение** расширяет семантику строительного блока (класса, объекта, пакета и т.д.), то есть добавляет новые правила или модифицирует существующие. Ограничения показывают как текстовую строку, заключенную в фигурные скобки {}. Может быть ограничение на несколько элементов.

2) **Теговая величина** (tagged value) расширяет характеристики строительного блока: позволяет создать новую информацию в спецификации конкретного элемента. Теговую величину показывают как строку вида: { имя теговой величины = значение }.

3) **Стереотип** расширяет словарь языка, т.е. позволяет создавать новые виды строительных блоков, которые являются производными от существующих, но учитывают специфику проблемы. Изображается стереотип в двойных угловых скобках <<>>. Стереотип может быть задан на любом предмете в модели, а также на отношении.

Профайл UML – ряд расширений мета-модели UML, указывающих, как элементы модели UML могут расширяться и настраиваться с применением стереотипов, ограничений и теговых величин. Профайл UML – это замкнутый набор таких расширений, определенный для какой-либо цели.

31. Диаграммы классов (точки зрения).

Диаграмма классов описывает классы (интерфейсы) и отражает отношения, существующие между ними. У диаграммы классов два главных элемента – **классы** (интерфейсы), а также **отношения между ними**. На значках класса обычно указывают важные в зависимости от точки зрения на диаграмму ***атрибуты*** и ***операции*** класса. Даже на диаграммах последнего уровня (точка зрения реализации) не перечисляются все атрибуты и операции (их может быть слишком много) Для этих целей служат спецификации классов, где можно подробно объявлять все элементы класса.

Иногда бывает необходимо ограничить количество экземпляров класса. По умолчанию предполагается, что на количество объектов нет ограничений. ***Множественностью*** класса называется количество его экземпляров. Она записывается в правом верхнем углу значка класса.

Точки зрения на диаграммы классов:

1) ***Концептуальная*** точка зрения – диаграмма классов описывает модель предметной области, в ней присутствуют только классы прикладных объектов;

2) Точка зрения ***спецификации*** – диаграмма классов применяется при проектировании информационных систем;

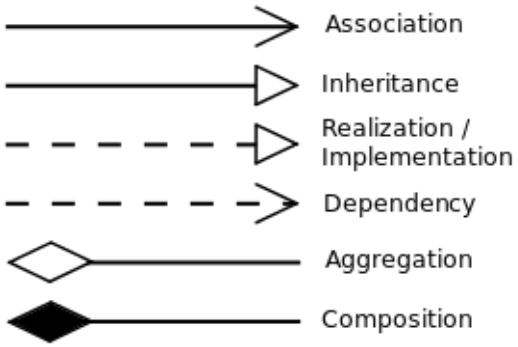
3) Точка зрения ***реализации*** – диаграмма классов содержит классы, используемые непосредственно в программном коде (при использовании ОО ЯП).

32. Отношения в диаграммах классов.

Классы в системе не существуют автономно, так как объекты одних классов взаимодействуют с объектами других (посылают друг другу сообщения). Поэтому говорят, что классы вступают в **отношения**.

В стандарте UML определены следующие типы отношений между классами (пакетами, интерфейсами):

- ассоциация (равноправные классы),
- обобщение (наследование),
- зависимость,
- реализация,
- агрегация (включение по ссылке),
- композиция (физическое включение).

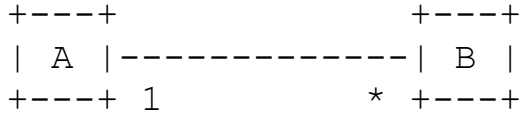


33. Отношения ассоциации «один ко-многим» и «многие ко-многим» в диаграммах классов этапа проектирования ПО.

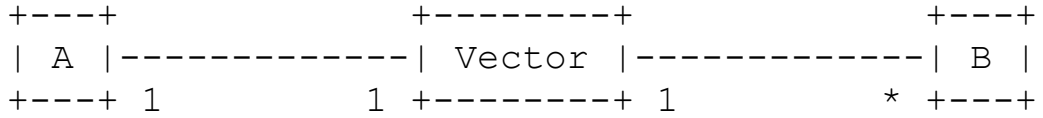
Ассоциация отображает структурные отношения между объектами классов. Это означает, что между объектами классов есть соединение, и через это соединение они могут посылать сообщения друг другу.

Любая ассоциация (и другие виды отношений) может показывать, сколько экземпляров одного класса взаимодействуют с помощью связи с одним экземпляром другого класса. Это называется заданием множественности связи. Множественность задается на концах линии связи.

Пример ***один-ко-многим***:



на самом деле это так:

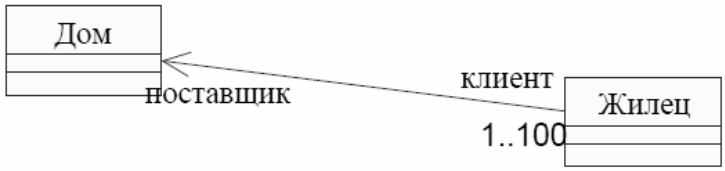


Пример ***многие-ко-многим***:

Как на самом деле – строится аналогично.

34. Отношение зависимости в диаграммах классов этапа проектирования. Стереотипы зависимости.

Это отношение показывает, что один класс ***ссылается*** на другой (зависит от другого). Таким образом, изменения во втором классе повлияют на первый. Отношение зависимости изображают пунктирной линией со стрелкой.



У отношения зависимости также как и у отношения ассоциации может быть задана ***множественность***.

Итак, зависимость является отношением использования между клиентом (зависимым элементом) и поставщиком (независимым элементом). Обычно зависимость означает, что операции клиента или вызывают операции поставщика, или имеют сигнатуры, в которых возвращаемое значение или аргументы принадлежат поставщику.

Отношение зависимости самое разнообразное среди всех типов отношений. В нотации UML предусмотрено 17 различных зависимостей. Для их различия задается **стереотип**, например, ***bind*** (подстановка параметров в шаблон), ***call*** (зависимость между операциями), ***instanceOf***, ***derive*** («может быть вычислен по»), ***instantiate*** (источник создает экземпляры целевого элемента).

35. Диаграммы вариантов использования, реализации вариантов использования.

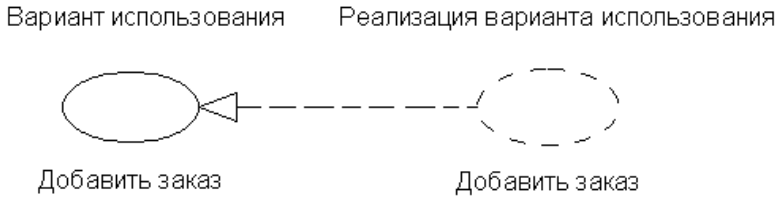
Диаграммы вариантов использования (UseCaseDiagram) определяют поведение системы с точки зрения пользователя. Поэтому они рассматриваются как главное средство на этапе определения и уточнения требований и основа общения между заказчиками и разработчиками. **Состав** диаграммы UseCase:

1) ***прецеденты*** (варианты использования). Описание последовательности действий, которые выполняются системой и производят для актера видимый результат. Поэтому каждый UseCase задает определенный путь использования системы, а набор всех элементов UseCase определяет полные функциональные возможности системы, т.е. границы системы.

2) ***актеры*** (действующие лица). Актеры представляют внешний мир, нуждающийся в работе системы. Варианты использования представляют действия, выполняемые системой в интересах актеров. Следует различать актеров и пользователей системы. Пользователь –это физический объект, который использует систему. Он может играть несколько ролей и поэтому может моделироваться несколькими актерами. Обратное тоже справедливо – одним актером могут быть разные пользователи. Актером может быть люди, внешняя система, время.

3) ***отношения между ними*** (связи). Между актером и прецедентом возможен только один тип отношения – *ассоциация*. Между актерами допустимо отношение *обобщения*. Между вариантами использования определены отношения *обобщения* и две разновидности отношения *зависимости* – включение и расширение.

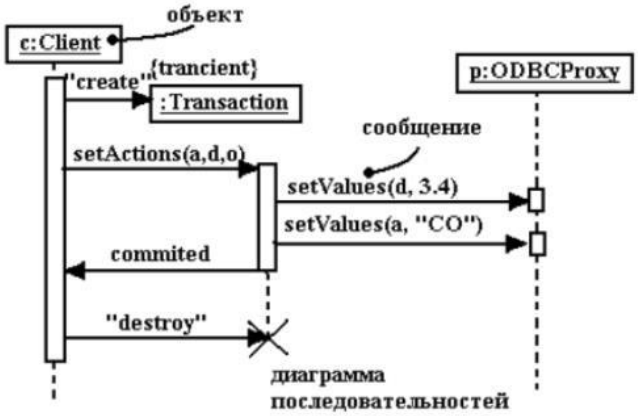
Реализации вариантов использования (или кооперации), как правило, фиксируются на Логическом уровне представления модели. Обозначения их соединяются отношением “реализация”. Кооперации содержат две составляющие – структурную (статическую) и поведенческую (динамическую). Статическая составляющая кооперации задается диаграммой классов (или несколькими диаграммами классов). Динамическая составляющая кооперации определяет поведение совместно работающих объектов и задается одной (или несколькими) диаграммами последовательности.



36. Диаграммы взаимодействий.

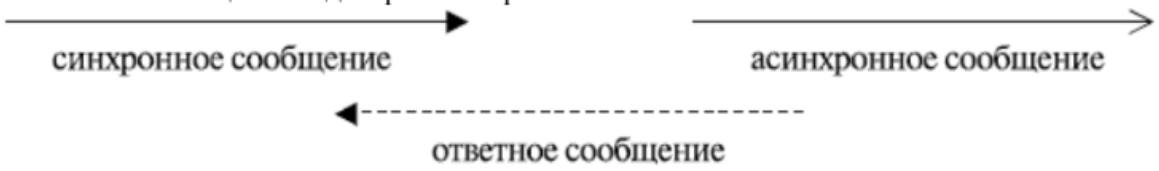
Диаграммы взаимодействий отражают взаимодействие объектов друг с другом. Они бывают двух видов: **диаграммы последовательностей** (SequenceDiagrams) и **диаграммы сотрудничества** (CollaborationDiagrams). Оба вида диаграмм служат для моделирования поведения объектов, т.е. с их помощью можно смоделировать динамические аспекты системы. При этом диаграмма последовательностей акцентирует внимание на временной упорядоченности сообщений, а диаграмма сотрудничества – на структурной организации посылающих и принимающих сообщения объектов. Обе диаграммы семантически эквивалентны, т.е. преобразуются друг в друга без потери информации.

Каждый прямоугольник в верхней части диаграммы – конкретный объект. Вертикальные линии представляют линии жизни объектов. Обмен сообщениями между объектами отображается с помощью горизонтальных линий, проведенных между соответствующими вертикальными линиями.



Диаграммы последовательностей характеризуются двумя особенностями, отличающими их от кооперативных диаграмм. Во-первых, на них показана ***линия жизни объекта***. Это вертикальная пунктирная линия, отражающая существование объекта во времени. Вторая особенность диаграмм последовательностей − ***фокус управления***. Он изображается в виде вытянутого прямоугольника, показывающего промежуток времени, в течение которого объект выполняет какое-либо действие, непосредственно или с помощью подчиненной процедуры.

Наиболее общую форму управления задает процедурный поток или поток синхронных сообщений. Процедурный поток рисуется стрелками с заполненными наконечниками. Все сообщения процедурной последовательности считаются синхронными.



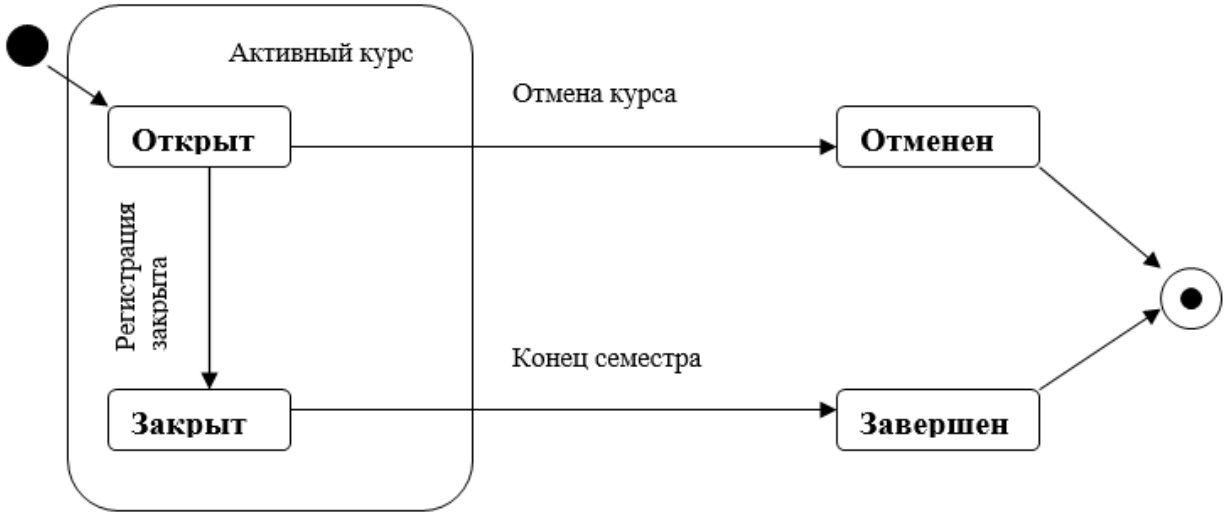
37. Диаграммы состояний.

На диаграмме состояний отображают жизненный цикл ***одного объекта*** (или класса объектов), начиная с момента его создания и заканчивая разрушением. Поэтому с помощью таких диаграмм удобно моделировать динамику поведения объекта класса.

Два основных элемента диаграммы состояний – это состояния и переходы между ними.

1) **Состоянием** называется одно из возможных условий, в которых может находиться объект класса между двумя событиями. На языке UML состояние изображают в виде прямоугольника с закругленными краями.

2) **Переходом** называется перемещение объекта из одного состояние в другое. Изображается в виде линии со стрелкой. Переходы могут быть рефлексивными: объект переходит в то же состояние, в котором он находится. На переходах можно записывать события и действия. Событие – определяет условие, когда переход может быть выполнен. Событием может быть объект (класс) или операция (чаще всего).



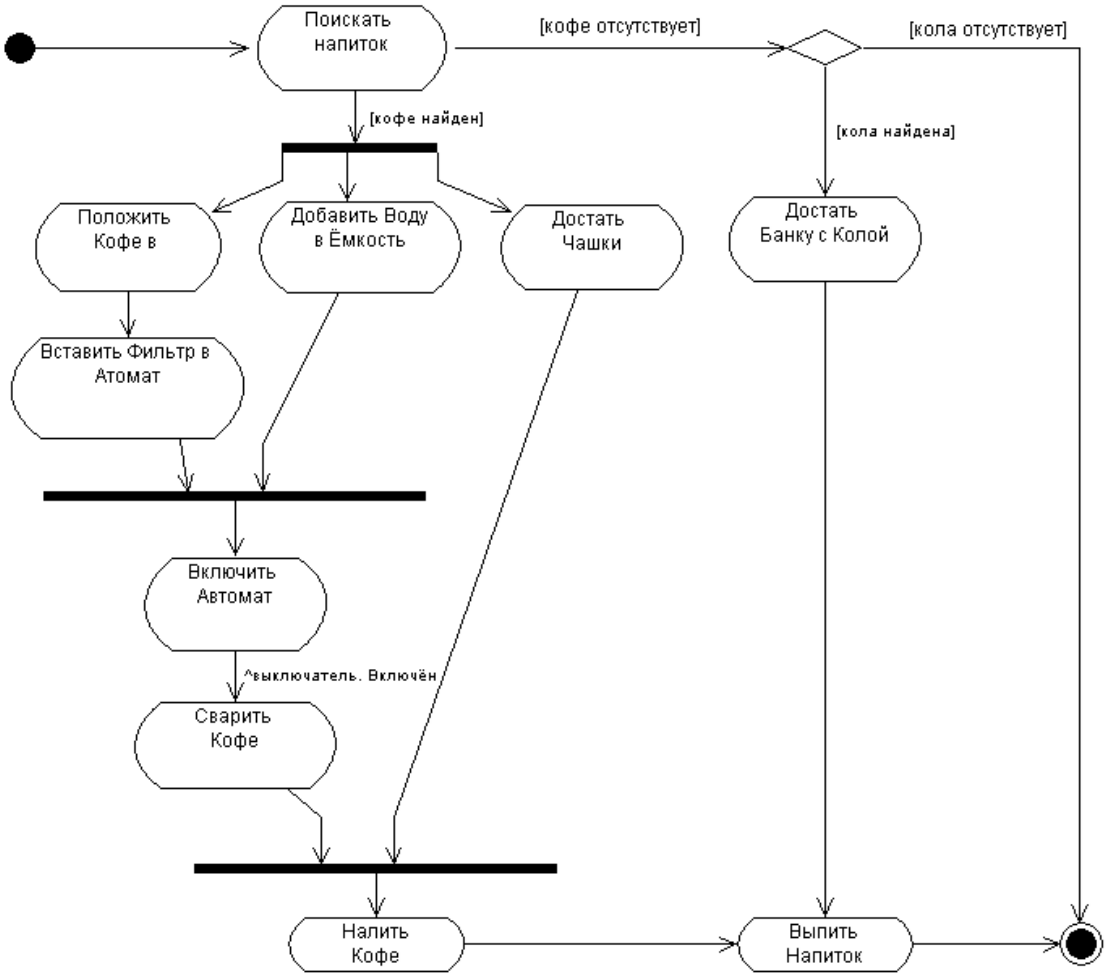
38. Диаграммы активности (деятельности).

Диаграммы деятельностей используются для описания *поведения* систем.

Основным элементом диаграммы деятельностей является ***деятельность***. Причем диаграммы деятельностей, как и диаграммы классов, могут строиться с *трех различных* точек зрения: с *концептуальной*, с точки зрения *спецификации* и с точки зрения *реализации*. В соответствии с точкой зрения деятельность рассматривается по-разному.

На концептуальной диаграмме *деятельность* – это некоторая *задача*, которую необходимо *автоматизировать* или выполнить *вручную*.

На диаграмме, построенной с точки зрения *спецификации* или *реализации*, *деятельность* – это некоторый *метод* над классом.



Различие между диаграммой деятельностей и блок-схемой в том, что блок-схемы показывают последовательные процессы, а диаграммы деятельностей могут поддерживать дополнительно параллельные процессы.

39. Каркасы и паттерны. Примеры.

Образец (**паттерн**) проектирования предлагает типичное решение типичной проблемы в определенном контексте. Образцы проектирования – это шаблоны взаимодействующих классов (объектов) и методов.

Наиболее известной из книг по паттернам является книга “Банды четырех” (Гамма, Хелм, Джонсон, Влиссидес), в которой описаны модели 23 паттернов.

Структурные паттерны (Адаптер, Компоновщик, Декоратор, Мост, Фасад и др.):

- используются с целью образования более крупных структур из классов и объектов для получения новой функциональности
- имеют дело со способами представления объектов (такими, как деревья или связные списки)
- позволяют пользоваться множеством объектов как единым целым
- позволяют организовать совместную работу нескольких библиотек, которые были разработаны отдельно друг от друга (Адаптер, Мост).

Порождающие (креационные) паттерны (Абстрактная Фабрика, Строитель, Фабричный метод, Прототип, Одиночка).

- важны, когда система больше зависит от композиции объектов, чем от наследования классов
- основной упор делается не на жесткое кодирование фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений.
- затем с помощью композиции можно получить более сложные объекты (например, лабиринты и деревья).

Паттерны поведения (Команда, Итератор, Интерпретатор, Посредник, Наблюдатель, Состояние, Шаблонный метод и др.):

- позволяют следить за поведением объектов (например, выдавать отчет о коллекции объектов в определенном порядке)
- связаны с распределением обязанностей между объектами, т.е. речь в них идет о типичных способах взаимодействия объектов.
- различают паттерны поведения уровня класса (используется наследование) и уровня объекта (используется композиция – как объекты получают информацию друг о друге).

Каркас – это больше, чем механизм (больше, чем образец). Он включает в себя много механизмов, которые работают совместно для решения типичной проблемы для конкретной предметной области. Поэтому с помощью каркаса можно описать “скелет” архитектуры со всеми управляющими классами. Акцент в каркасе делается на повторном использовании дизайна, а не кода (как в библиотеках). Типичным примером каркаса является API язык Java (Application Programming Interface) и MFC (библиотека фундаментальных классов C++).

Основные ***различия паттернов и каркасов***:

1. Паттерны проектирования более абстрактны, чем каркасы. Паттерн необходимо реализовать всякий раз, когда он используется.

2. Каркасы описывают структуру и поведение системы в целом, а паттерны описывают структуру и поведение сообщества классов.

3. Паттерны мельче, чем каркасы (типичный каркас содержит несколько паттернов).

4. Паттерны менее специализированы, чем каркасы. Каркас почти всегда создается для конкретной предметной области, а паттерны можно использовать в приложениях почти любого вида (хотя есть и более специализированные паттерны).

40. Основные понятия и определения теории тестирования. Подходы к тестированию. Стратегии тестирования. Критерии тестирования.

Основные понятия:

Тестирование программного обеспечения (software testing) – это процесс анализа или эксплуатации программного обеспечения с целью выявления дефектов.

Дефект (баг) – это несоответствие требованиям к программному продукту или его функциональной спецификации.

Ожидаемый результат – это предполагаемое корректное поведение системы (программного продукта). Если реальное поведение системы, которое мы наблюдаем, не совпадает с тем, что мы ожидали увидеть, мы можем говорить о том, что имеет место дефект.

Test Case (тестовый случай) – набор тестовых данных, условий выполнения теста и последовательность действий тестировщика, а также ожидаемый результат, которые разрабатывается с целью проверки тех или иных аспектов работы программы.

Тестовый план – часть проектной и тестовой документации, описывающий что, когда, кем, и как будет тестироваться.

Build (билд) – это промежуточная версия программного продукта, которая поставляется разработчиками для тестирования.

Подходы к тестированию:

- ***статическое тестирование*** (процесс, связанный с анализом разработки программного обеспечения) – предусматривает проверку любых рабочих продуктов, например, таких как программный код, требования к программному продукту, функциональная спецификация, архитектура, дизайн и т.д. Статическое тестирование по существу есть все, что можно сделать для выявления дефектов без прогона программного кода.
- ***динамическое тестирование*** – тестовая деятельность, предусматривающая эксплуатацию программного продукта. Динамическое тестирование состоит из запуска программы, прогона всех ее функциональных модулей и сравнения ее фактического поведения с ожидаемым, используя пользовательский интерфейс.

Стратегии тестирования:

- метод ***белого (/прозрачного/стеклянного) ящика*** – тестирование программного кода без его непосредственного запуска. Применяется при *структурном тестировании*. Его тесты основаны на знании кода приложения и его внутренних механизмов. Цель – проверить каждый из его компонентов, модулей, процедур и подпрограмм → *компонентное (модульное) тестирование (unit testing)*.
- метод ***черного ящика*** – тестирование исходя из знаний функциональных требований к тестируемому продукту. Тестировщик тестирует программу так, как с ней будет работать конечный пользователь, и он ничего не знает о внутренних механизмах и алгоритмах, по которым работает код программы. Цель – проверить работу всех функций приложения на соответствие функциональным требованиям → *функциональное тестирование*.

41. Критерии тестирования стратегии «черного ящика».

Метод черного ящика – тестирование исходя из знаний функциональных требований к тестируемому продукту, используется для тестирования программы при ее запуске на исполнение. Тестировщик тестирует программу так, как с ней будет работать конечный пользователь, и он ничего не знает о внутренних механизмах и алгоритмах, по которым работает код программы. Цель – проверить работу всех функций приложения на соответствие функциональным требованиям.

1) ***Эквивалентное разбиение.***
   a. Выделение классов эквивалентности. Путем рассмотрения каждой входной области данных программы и разбиении её на ≥2 группы. Класс эквивалентности – набор данных с общими свойствами. Программы, обрабатывая разные элементы одного класса эквивалентности, должны вести себя одинаково. Два вида входных классов эквивалентности: допустимые, недопустимые данные;
   b. Построение тестов. Надо чтобы все входные классы эквивалентности были покрыты.

2) ***Анализ граничных значений/условий.*** Граничные условия – это ситуации, которые возникают на границе классов эквивалентности. Анализ граничных условий отличается от эквивалентного разбиения следующими моментами:
   a. Выбор элемента в классе эквивалентности осуществляется не произвольно, а так, чтобы проверить каждую границу класса;
   b. Рассматриваются не только входные классы эквивалентности, но и правильные выходные классы эквивалентности.

3) ***Метод функциональных диаграмм.*** Делается семантический анализ внешних спецификаций, а затем осуществляется перевод их на язык логических отношений для каждой реализуемой функции. Такие преобразования представляются в форме логической диаграммы, которая и называется функциональной диаграммой.

42. Критерии тестирования стратегии «белого ящика».

Метод белого (/прозрачного/стеклянного) ящика – тестирование программного кода без его непосредственного запуска. Применяется при структурном тестировании. Его тесты основаны на знании кода приложения и его внутренних механизмов. Цель – проверить каждый из его компонентов, модулей, процедур и подпрограмм.

1) ***Покрытие операторов***: тестов должно быть столько, чтобы каждый оператор программы выполнялся хотя бы один раз.

2) ***Покрытие решений***, ***условий***, ***решений+условий***. Решение – логическое выражение в условных операторах и операторах цикла. Условие – отдельное логическое условие в решении. Тесты должны выбираться таким образом, чтобы каждое решение (условие) принимало на этих тестах значение true, false по крайней мере один раз. Критерий покрытия условий не всегда удовлетворяет критерию покрытия решений.

3) ***Покрытие путей***. Этот критерий требует столько тестов, чтобы покрыть все пути в программе хотя бы один раз. Это очень сильный критерий, но в практике тестирования его обычно не применяют из-за большого числа различных путей в программе.

43. Способы тестирования программ, состоящих из модулей (классов, блоков).

Модульное тестирование (Unit testing) – позволяет проверить функционирование отдельно взятого элемента системы. Модулем может быть отдельно взятая функция или набор функций, отдельно взятый класс или набор классов, компонент, выполняющий какую-то функциональность и имеющий или чаще не имеющий пользовательского интерфейса.

Интеграционное тестирование (Integration testing) – процесс проверки взаимодействия между программными компонентами/модулями. Классические стратегии интеграционного тестирования (используются для иерархически структурированных систем):

- “сверху-вниз” (модули объединяются движением сверху вниз по иерархии, начиная от главного управляющего модуля);
- “снизу-вверх” (сборка и тестирование системы начинаются с модулей, которые располагаются на нижних уровнях иерархии).

К моменту выполнения интеграционного тестирования должны быть проверены отдельные модули, взаимодействие которых мы собираемся проверять.

Интеграционное тестирование может делиться на:

- *интеграционное юнит-тестирование* (взаимодействие объектов на уровне функций и классов);
- *интеграционное тестирование приложения* (взаимодействие различных частей приложения).

44. Классический процесс тестирования ПО (методика тестирования).

Классический процесс тестирования обеспечивает проверку результатов, полученных на каждом этапе разработки. Вначале проверяются простые модули, затем модули объединяются в программу (систему). Тестирование состоит в проверке на соответствие программного продукта требованиям заказчика, а также осуществляется проверка взаимодействия ПО с другими компонентами компьютерной системы.

Методика тестирования ПС:

1) ***Тестирование элементов (модулей)***. Индивидуальная проверка каждого модуля, проверяются результаты этапа кодирования.

2) ***Тестирование интеграции***. Тестирование сборки модулей в ПС, выявляются ошибки этапа проектирования ПС. В основном применяются критерии стратегии «черного ящика»

3) ***Тестирование правильности (функциональное)***. Проверка реализации в ПС всех функциональных и поведенческих требований, а также проверка требований эффективности, проверяется корректность этапа анализа требований. Используются исключительно критерии стратегии «черного ящика».

4) ***Системное тестирование***. Проверка правильности объединения и взаимодействия всех элементов компьютерной системы, а также проверка реализации всех системных функций, выявляются дефекты этапа системного анализа. Используются особые типы системных тестов: тестирование восстановления; тестирование безопасности; тестирование производительности и стрессовое тестирование.

45. Модульное тестирование. Тестирование интеграции.

Модульное тестирование (Unit testing) – позволяет проверить функционирование отдельно взятого элемента системы. Модулем может быть отдельно взятая функция или набор функций, отдельно взятый класс или набор классов, компонент, выполняющий какую-то функциональность и имеющий или чаще не имеющий пользовательского интерфейса.

Интеграционное тестирование (Integration testing) – процесс проверки взаимодействия между программными компонентами/модулями. Классические стратегии интеграционного тестирования (используются для иерархически структурированных систем):

- “сверху-вниз” (модули объединяются движением сверху вниз по иерархии, начиная от главного управляющего модуля);
- “снизу-вверх” (сборка и тестирование системы начинаются с модулей, которые располагаются на нижних уровнях иерархии).

К моменту выполнения интеграционного тестирования, как правило, должны быть проверены отдельные модули, взаимодействие которых мы собираемся проверять.

Интеграционное тестирование может делиться на:

- *интеграционное юнит-тестирование* (взаимодействие объектов на уровне функций и классов)
- *интеграционное тестирование приложения* (взаимодействие различных частей приложения).

46. Функциональное тестирование. Уровни функционального тестирования.

Цель – определить, соответствует ли приложение предъявляемым к нему требованиям. Подтверждение правильности ПС выполняется с помощью тестов ***«черного ящика»***.

Во многих компаниях тестирование разделено на **три уровня** (приемочный, критический и расширенный тесты), отличающиеся между собой объемом проводимого тестирования и различными вариантами использования программного продукта.

Приемочный тест (Smoke test) – самый первый и короткий тест, проверяющий работу основной функциональности программного продукта. Данный тест длится от получаса до 2-3-х часов максимум в зависимости сложности программы, по результатам которого принимается решение о целесообразности дальнейшего тестирования. Если программа не прошла приемочный тест, она отправляется на доработку к программистам.

Критический тест (Critical path test) – основной вид теста, во время которого проверяются основная функциональность программного продукта критичная для конечного пользователя, при стандартном его использовании. В рамках данного тестирования, как правило, проверяется большинство требований предъявляемых к программному продукту.

Расширенный тест (Extended test) – углубленный тест, при котором проверяется нестандартное использование программного продукта. Прогоняются различные сложные, логически запутанные сценарии, совершаются действия, которые конечный пользователь будет совершать очень редко. Программа должна корректно реагировать на любые, даже самые случайные, действия пользователя и работать надежно в любой ситуации.

47. Системное тестирование. Основные виды.

Системное тестирование – проверка правильности объединения и взаимодействия всех элементов компьютерной системы, а также проверка реализации всех системных функций, выявляются дефекты этапа системного анализа.

Виды системных тестов:

- тестирование ***восстановления*** (ПС должны восстанавливаться после отказов и продолжать обработку в пределах заданного времени, система должна быть отказоустойчивой);
- тестирование ***безопасности***;
- тестирование ***производительности***;
- ***стрессовое*** тестирование (навязывание программе ненормальных ситуаций, называется "расшатать" систему).

48. Особенности тестирования объектно-ориентированных программ.

Основные отличия процесса тестирования объектных программ от традиционных (процедурных) состоят в том, что аналитические модели непосредственно отображаются на проектные модели, которые в свою очередь отображаются на программы. Следовательно, можно ***начинать тестирование на стадии анализа***, затем тестировать ***проект*** и доводить тестирование на этапе ***реализации***.

На практике следуют такому порядку:

- Тестирование моделей (аналитических и проектных).
- Тестирование классов (вместо тестирования модулей).
- Тестирование взаимодействий и функционирования компонентов.
- Тестирование подсистем и систем.
- Приемочные испытания.

Основной метод здесь – метод *целенаправленной проверки* (инспекции). Для этой цели используются следующие виды диаграмм: ***диаграммы Use Case*** с текстовым описанием каждого варианта использования; ***диаграммы последовательностей*** для сценариев каждого варианта использования, ***диаграммы классов анализа*** и проектные диаграммы (описывают архитектуру ПО); ***диаграммы состояний*** объектов класса и ***диаграммы действий***.

49. Тестирование классов.

Тестирование классов – ***проверка реализации класса на соответствие спецификации*** этого класса. Прежде чем приступить к тестированию того или иного класса, потребуется определить: будем тестировать его в автономном режиме как ***модуль*** или как более крупный ***компонент системы***. Для этого необходимо учитывать следующие факторы:

- Роль этого класса в системе (в частности, степень связанного с ним риска).
- Сложность класса (может измеряться количеством операций, состояний или числом связей с другими классами).
- Объем трудозатрат, которые необходимы для разработки драйвера для тестирования класса.

Тестирование класса должно проводиться до того, как возникнет необходимость использования этого класса в других компонентах ПО. И каждый раз, когда меняется что-то в реализации класса, должно проводиться ***регрессионное тестирование*** этого класса.

Тестовый драйвер должен создавать экземпляр классов и окружать эти экземпляры соответствующей средой, чтобы была возможность прогона (запуска) тестов. Драйвер обычно посылает одно или большее число сообщений объекту класса, затем проверяет исход этих сообщений. В обязанности драйвера входит удаление любого созданного им экземпляра.

На практике широко используются 3 ***критерия построения тестов для классов***:

1) покрытие, ориентированное на состояния (диаграмма состояний);

2) покрытие, ориентированное на ограничения (пред- и постусловия операций);

3) покрытие программного кода (любой критерий «белого» ящика).

При тестировании некоторых операций класса могут использоваться любые критерии стратегии «черного» ящика (анализ граничных условий, например).

50. Тестирование взаимодействия классов и функционирования компонентов.

У большинства классов имеются партнеры по сотрудничеству, т. е. методы такого класса ***взаимодействуют с экземплярами других классов***.

Основное назначение тестирования взаимодействий состоит в том, чтобы убедиться, что происходит ***правильный обмен сообщений между объектами***, классы которых прошли тестирование в автономном режиме.

Выявить такие классы можно на ***диаграмме классов*** (или по спецификации классов). Процесс тестирования взаимодействий классов заметно облегчается, если число связей между ними невелико. Тогда не придется создавать много программ-драйверов.

Для выявления всех связей классов особенно полезны диаграммы классов и диаграммы последовательностей. Они показывают, каким классам (объектам) нужен тестируемый класс, т.е. кто использует заявленные операции класса. И наоборот, к операциям каких классов обращаются объекты тестируемого класса.

51. Тестирование подсистем и систем. Приемочные испытания.

Если определены спецификации **подсистем и систем** (должны быть отражены прежде всего в диаграммах Use Case и диаграммах взаимодействий для каждого варианта использования), то в этом случае уже можно готовить ***системные тесты***. Но системное тестирование предполагает ***тестирование готового приложения***, в котором реализованы все (или часть) функций и которое должно функционировать в определенной среде.

Тестовые наборы должны ***проверять все сценарии*** для всех реализованных вариантов использования. Поэтому план тестирования системы представляет собой более формальный и довольно объемный документ.

Приемочные испытания устраиваются заказчиком (или спонсором) проекта до официального окончания разработки.

Этот вид тестирования проводится в среде разработки на площадке заказчика. Заказчик определяет, удовлетворительно ли работает программный продукт с его точки зрения. Вообще говоря, приемочные испытания представляют собой специальный вид системного тестирования.

52. Основы автоматизированного тестирования. Преимущества автоматизации, области применения, оценка целесообразности.

Для тестирования ПО смысл автоматизации – в повышении эффективности работы.

Цели автоматизации:

- ***приемочные тесты*** (проводятся для каждого build приложения, а зачастую и на нескольких конфигурациях),
- ***регрессионные тесты*** (набор тестов, который нужно повторять снова и снова, для того чтобы быть уверенным, что приложение работает как нужно, несмотря на вносимые изменения)
- тесты, которые ***не могут быть выполнены вручную*** в приемлемые сроки.

Преимущества автоматизации:

- Скорость выполнения по сравнению с ручным тестированием значительно выше;
- Есть возможность запуска тестов в нерабочее время;
- Возможность повтора тестов, причем в точности так, так как было задумано, и с теми же данными;
- Исключение ошибок (человеку свойственно ошибаться);
- Накопление результатов в формализованном виде;
- Освобождение тестировщиков от регулярного выполнения одних и тех же, простых по сути, но требующих повышенного внимания тестов, и возможность заняться более интересной и креативной работой;
- Управление несколькими машинами одновременно – распределенный тест;
- Взаимодействие с базами данных "напрямую";
- Перезагрузка машины в другую ОС и вход в систему под разными пользователями;
- Поиск "поломанных" ссылок на веб-страницах;
- Создание теста на одном браузере и его запуск на другом без изменений.

Оценка целесообразности. При оценке затрат (подразумевается что тесты для ручного тестирования уже разработаны и документированы) нужно лишь учитывать следующие величины:

- Время на проход теста вручную
- Количество таких проходов
- Время на разработку и отладку автоматического теста
- Время его работы
- Время на исправление скрипта при изменениях в приложении.

53. Методы автоматизированного тестирования.

1. **Метод функциональной декомпозиции**

При разработке скриптов все тестовые случаи приводятся к некоторым более ***фундаментальным задачам***, которые включают:

- Навигацию (например, доступ к странице определения заказа из главного меню).
- Специфические (бизнес) функции (например, оформление заказа).
- Проверку данных (например, проверка состояния заказа).
- Возврат к навигации (возврат к главному меню).

2. **Метод Data-driven**

Этот метод является продолжением предыдущего, и в нем параметры функций (по сути данные для тестов) ***выносятся за пределы кода*** скрипта.

3. **Метод Keyword-driven**

Этот метод использует документы, являющиеся тестовыми случаями либо набором тестовых случаев – ***тестовыми сценариями***. Тестовые сценарии описывают действия, которые необходимо выполнить, наборы входных данных и ожидаемый результат, и представляют собой электронную таблицу, содержащую специальные ключевые слова (keywords). Ключевые слова описывают процесс выполнения тестовых случаев.