ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ ‘18

1. История развития языков программирования высокого уровня

Тенденции:

1) Перемещение акцентов от программирования отдельных деталей к программированию более крупных компонент;
2) Развитие и совершенствование ЯП высокого уровня;
3) Рост сложности;
4) Переход от императивных языков (указывают компьютеру, что делать) к декларативным (описывают ключевые абстракции предметной области);

Поколения ЯП по языковым конструкция (Вэгнер):

1) **Первое поколение** (1954 – 1958: **программирование мат. формул**) FORTRAN I, ALGOL-58, Flowmatic: математические формулы.
2) **Второе поколение** (1959 – 1961: **алгоритмические абстракции**) FORTRAN II (Подпрограммы), ALGOL-60 (Блочная структура, типы данных), COBOL (Описание данных, работа с файлами), Lisp (Списки, указатели, сборка мусора)
3) **Третье поколение** (1962 – 1970: **введение типов данных**) PL/I (FORTRAN + COBOL + ALGOL), ALGOL-68, Pascal, Simula (Классы, абстрактные данные)
4) **Потерянное поколение** (1970 – 1980: **объектные и объетно-ориентированные**) Smalltalk, Afa, C++. Основным элементом конструкции в указанных языках служит модуль, составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.

2. Архитектура ЯП (три поколения)

Поколения ЯП по языковым конструкция (Вэгнер):

1) **Первое поколение** (1954 – 1958: **программирование мат. формул, научно-инженерные применения**) FORTRAN I, ALGOL-58, Flowmatic: математические формулы.
2) **Второе поколение** (1959 – 1961: **алгоритмические абстракции**) FORTRAN II (Подпрограммы), ALGOL-60 (Блочная структура, типы данных), COBOL (Описание данных, работа с файлами), Lisp (Списки, указатели, сборка мусора)
3) **Третье поколение** (1962 – 1970: **введение типов данных**) PL/I (FORTRAN + COBOL + ALGOL), ALGOL-68, Pascal, Simula (Классы, абстрактные данные)
4) **Потерянное поколение** (1970 – 1980: **объектные и объетно-ориентированные**) Smalltalk, Afa, C++. Основным элементом конструкции в указанных языках служит модуль, составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.

Топология (*основные элементы языка программирования и их взаимодействие*) языков 1 - начала 2 поколений: программы имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм. Нет разделения данных, много перекрестных связей между подпрограммами.

Топология языков конца 2 – начала 3 поколений: рост алгоритмических абстракций: механизмы передачи параметров, структуры управления и области видимости, методы структурного проектирования.

Топология языков конца 3 поколения: появление отдельно компилируемых модулей, собственные типы данных.

3. Архитектура объектно-ориентированных ЯП

Сложность задачи часто обусловнлена 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, которая воплощается в таких языках, как Pascal. Естественным завершением реализации этих идей, начавшейся с языка Simula и развитой в последующих языках в 1970-1980-е годы, стало сравнительно недавнее появление таких языков, как Smalltalk, Object Pascal, C++, CLOS, Ada.

Эти языки получили название **объектных** или **объектно-ориентированных**. Основным элементом конструкции в указанных языках служит модуль, составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.

По этой же причине структура программ малой и средней сложности при объектно-ориентированном подходе представляется графом, а не деревом, как в случае алгоритмических языков. Кроме того, уменьшена или отсутствует область глобальных данных. Данные и действия организуются теперь таким образом, что основными логическими строительными блоками наших систем становятся классы и объекты, а не алгоритмы.

Кластеры абстракций в больших системах могут представляться в виде многослойной структуры. На каждом уровне можно выделить группы объектов, тесно взаимодействующих для решения задачи более высокого уровня абстракции. Внутри каждого кластера мы неизбежно найдем такое же множество взаимодействующих абстракций.

4. Сложность, присущая ПО, причины. Проблемы при создании сложных систем

Нас интересует разработка того, что мы будем называть промышленными программными продуктами. Системы подобного типа обычно имеют большие время жизни и большое количество пользователей оказывается в зависимости от их нормального функционирования.

Сложность вызывается четырьмя основными причинами:

1) сложность **реальной предметной области (проблемы)**, из которой исходит заказ на разработку. Кроме того, большие системы имеют тенденции к **эволюции** (*внесение изменений в систему в ответ на изменившиеся требования*) в процессе из использования, возникает необходимость **сопровождения** (*устранение ошибок*) ПО, **сохранения** (*поддержание жизни системы*)
2) сложность **управления процессом разработки**;
3) сложность обеспечения достаточную **гибкость программы**;
4) сложность описания поведения **больших дискретных систем**.(*много переменных и потоков управления*)

Последствия неограниченной сложности. Кризис прогрмамного обеспечения: "Чем сложнее система, тем легче ее полностью развалить". Неумение создавать сложные ПС проявляется в проектах, которые выходят за рамки установленных сроков и бюджетов и к тому же не соответствуют начальным требованиям.

5. Структура сложных систем (5 признаков). Примеры сложных систем.

Пять общих признаков сложных систем (СС):

1) СС часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы.
2) Выбор элементарных компонент относительно произволен.
3) Внутрикомпонентная связь обычно сильнее, чем связь между компонентами.
4) Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.
5) СС – результат развития более простой системы.

Структура ПК (*умеренная сложность*)

Элементы: системная плата, монитор, клавиатура, устройства внешней памяти. Каждый из них можно разделить на составляющие. Системная плата: оперативная память, центральный процессор, шина. Центральный процессор: регистры и схемы управления, которые сами состоят из более простых деталей: диодов, транзисторов и т.д.

В этой системе различные уровни абстракции могут быть рассмотрены отдельно. На каждом уровне абстракции мы находим набор устройств, которые совместно обеспечивают некоторые функции более высокого уровня, и выбираем уровень абстракции, исходя из наших специфических потребностей.

Структура растений и животных

Растение: корни, стебли, листья. Все части на одном уровне абстракции взаимодействуют вполне определенным образом. Например, на высшем уровне абстракции, корни отвечают за поглощение из почвы воды и минеральных веществ. Корни взаимодействуют со стеблями, которые передают эти вещества листьям. Листья в свою очередь используют воду и минеральные вещества, доставляемые стеблями, и производят при помощи фотосинтеза необходимые элементы.

Для каждого уровня абстракции всегда четко разграничено "внешнее" и "внутреннее": существует четкое разделение их функций.

«Унифицированные единицы». Клетки служат основными строительными блоками всех структур растения. Существует огромное количество разнообразных клеток: содержащие и не содержащие хлоропласт, клетки с проницаемое и непроницаемой для воды оболочкой, живые и умершие клетки.

Фактически не существует централизованных частей, которые непосредственно координируют деятельность более низких уровней. Вместо этого отдельные части являются посредниками посредники, каждый из которых ведет себя достаточно сложно и при этом согласованно с более высокими уровнями. Только благодаря совместным действиям большого числа посредников образуется более высокий уровень функционирования растения. Наука о сложности называет это **возникающим поведением**. Поведение целого сложнее, чем поведение суммы его составляющих.

6. Типовая и структурная иерархия в объектно-ориентированной методологии

Каноническая форма сложной системы. Обнаружение общих абстракций и механизмов значительно облегчает понимание сложных систем. Существует два типа иерархий: **структурная** («*быть частью*») и **типовая** («*is a*»), их также называют **структурами классов** и **объектов** соответственно.

Объединяя понятия структуры классов и структуры объектов с пятью признаками сложных систем, мы приходим к тому, что все СС можно представить в канонической форме. Каждая иерархия является многоуровневой, причем в ней классы и объекты более высокого уровня построены из более простых. Какой класс или объект выбран в качестве элементарного, 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. С введением структуры классов мы размещаем в ней общие свойства экземпляров.

Наиболее успешны те ПС, в которых заложены хорошо продуманные структуры классов и объектов и которые обладают пятью признаками сложных систем. Структуры классов и объектов системы мы будем называть **архитектурой системы**.

Один из способов управления СС: «*разделяй и властвуй*». При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. *Декомпозиция* вызвана сложностью программирования системы, поскольку именно эта сложность вынуждает делить ее пространство состояний. Существует два типа декомпозиции: **алгоритмическая** декомпозиция (*концентрирует внимание на порядке происходящих событий*) и

объектноориентированная декомпозиция (*придает особое значение агентам, которые являются либо объектами, либо субъектами действия*).

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

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

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.

Под технологией программирования понимается наука об оптимальных способах проведения процесса программирования. ТП должна охватывать содержание процесса разработки ПС на всех его этапах, включая модификацию и сопровождение ПС.

ПО – это группа взаимосвязанных и взаимодействующих модулей или программ.

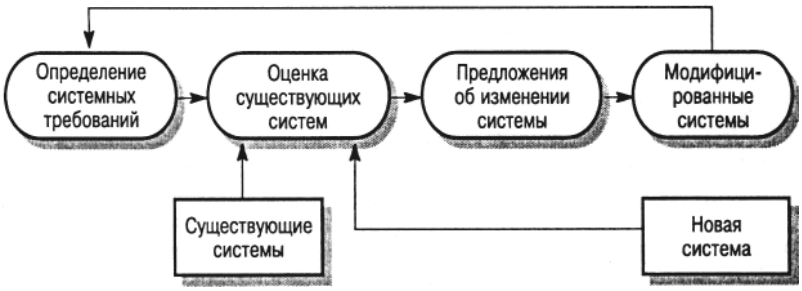
Программный продукт – это программа (или ПО), которой любой желающий может воспользоваться, которая надежна и эффективна.
Программное изделие – это программный продукт, который удовлетворяет определенному стандарту.
Программная система – программный продукт + что-то еще. Что это? Может быть, это - соответствующая платформа, OS, сервисное ПО.
Пакет прикладных программ – это программный комплекс, который предназначен для автоматизации решения определенного класса задач, т.е. еще говорят, что ППП должен покрывать конкретную предметную область. Покрытие области означает, что для любой задачи в данной области может быть построена программа (или совокупность программ), которая решает задачи.

8. Эволюция программного продукта. 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.



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

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). К материалам для модуляризации следует отнести:
   a. Тексты программ на языках программирования.
   b. Тексты указаний, применяемых к ПО
   c. Сопроводительная документация
   d. Проект системы (диаграммы, схемы, рисунки, тексты и т.д.)

Модуль – это выделенная часть первичного программного фонда (Майерс). Поэтому для различных первичных материалов и с различных точек зрения интерес представляют мотивы выделения модулей, а значит, и различные формы модулей.

При использовании различных методов проектирования (и ЯП, соответственно) модулем могут называться различные части исходного текста.

1) При использовании традиционных методологий и методов SD и DD наиболее удачным, на мой взгляд, является определение Фокса. **Модуль** – это относительно независимая часть программы (процедура, функция, подпрограмма и т.п.), которая имеет один вход и один выход, имеет небольшой размер, и которая выполняет одну-две функции.
2) При использовании объектно-ориентированного метода разработки ПС модулем раньше называли отдельный файл (Г.Буч). Еще раньше – файл, который можно отдельно транслировать (OPascal, Modula). Но с появлением новых объектно-ориентированных языков (C++, Java и т.д.) в файлах могут быть различные структурные единицы. Например, классы целиком, только их определения, могут быть интерфейсы как самостоятельные части, а 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.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.

CASE (Computer Aided Software Engineering) – технологии (средства, инструменты) – это совокупность методических материалов, автоматизированных методов и инструментальных средств, применяемых для разработки ПО как пригодного к коммерческому распространению продукта. Эти средства должны обеспечивать заданные экономические показатели процесса создания ПО, а также показатели эффективности и качества конечного программного продукта. Общая цель всех CASE-средств – поддержка любых процессов (на всех этапах) создания ПО.

Эволюция CASE–средств

Выделяют 5 периодов в программотехнике, которые характеризуются использованием инструментальных средств, таких как:

1) ассемблеров, дампов памяти;
2) компиляторов, трассировщиков;
3) символических отладчиков, пакетов программ;
4) систем анализа и управления исходными текстами;
5) CASE–систем.

На сегодняшний день характерно использование CASE-средств двух поколений: CASE-I и CASE-II.

CASE-I: первое поколение CASE явилось первой технологией, которая была адресована системным аналитикам и предназначалась для роста производительности труда, улучшения качества программ, а также для обеспечения контроля над процессом разработки. Главная цель CASE-1 состояла в отделении процесса проектирования ПО от его кодирования. Поэтому и развитие этого поколения CASE шло по следующим линиям:

1) использование методологии структурного проектирования ПО;
2) разработка средств моделирования данных и структур баз данных;
3) создание средств проектирования спецификаций ПО.

Однако для этого поколения CASE характерна *ограниченность функциональных возможностей*, а также *слабая интеграция* их средствами разработки программ (средства кодирования, тестирования, отладки и т.д.). Это как раз и стимулировало развитие CASE-средств второго поколения.

CASE-II направлены на автоматизацию процессов всего ЖЦ ПО. Все компоненты CASE-II могут быть разделены на несколько функциональных групп, решающих определенные задачи, например:

1) планирование проекта;
2) изучение возможностей решения проблемы;
3) определение требований;
4) системное проектирование;
5) программирование;
6) тестирование и отладка ПО;
7) измерение качества;
8) поддержка документирования;
9) управление процессом проектирования;
10) сопровождение ПО.

Классификация CASE-средств

Первая классификация (по функциональному ориентированию) постоянно расширяется. Тем не менее, есть возможность выделить следующее:

1) стратегическое планирование прикладных систем;
2) управление в поддержка полного ЖЦ ПО;
3) управление качеством ПО;
4) управление проектированием;
5) анализ и проектирование ПО;
6) генерация кода;-тестирование и отладка;
7) словари данных;
8) создание прототипов;
9) объектно-ориентированная обработка;
10) повторная разработка ПО.

Вторая классификация CASE-средств отражает функциональную наполненность. Здесь CASE-средства делятся на категории (tools, toolkit, workbench), типы (верхний, средний, нижний) и виды.

6 наиболее важных характеристик CASE средств:
1) поддержку полного ЖЦ ПО и полноту интеграции;
2) возможность создания прототипов;
3) наличие центральной БД проекта (repository);
4) использование стандартных средств и методов;
5) открытость архитектуры CASE;
6) простота использования в сочетания с мощными функциями.

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

Автоматизированные системы освобождают разработчика от трудоемких (рутинных) процессов. Далее, есть задачи, которые АС решают хорошо, и есть задача, которые они не способны решать в принципе. Например, если вы на диаграмме объектов показали, что один объект посылает сообщение другому объекту, АС может подтвердить, что это сообщение является частью протокола между объектами (пример автоматического контроля). АС может определить, например, что некоторые объекты класса никогда не используются. Однако АС не в состоянии определить новый класс (объект), чтобы упростить структуру классов.

Поскольку OOD выдвигает на первый план понятия ключевых абстракций и механизмов (отражаются в диаграммах классов в объектов), то необходимы инструменты, содержащие соответствующий набор условных обозначений.

Буч выделяет 5 различных типов инструментальных средств, которые оказываются полезными при OOD.

1) Первый тип составляют **графические АС со встроенными элементами нотаций**. Подобные инструменты используются, прежде всего, на самых ранних этапах процесса разработки ПО (анализ требований и проектирование начальной стадии), но на протяжении всего цикла разработки тоже пригодятся. Многие из инструментов этого типа позволяют осуществлять "обратную связь" при проектировании ОО системы, т.е. восстанавливать по исходному тексту структуру классов и модулей проектов.
2) Вторым типом инструментов, важных при OOD, являются инструменты **просмотра структуры классов** и **модулей архитектуры** системы. При просмотре фрагмента программы разработчику может понадобиться найти определение класса, которому принадлежат тот или иной объект. Подобные возможности имеются в окружении почти всех ОО языков программирования (системы проектирования).
3) Третий тип инструментов - это **пошаговые трансляторы** (обеспечивают трансляцию отдельных операторов и функций). Пошагово-транслируемые методы и пошагово-транслируемые описатели необходимы для отладки. Такого типа инструменты имеются, как правило, в системах программирования.
4) Четвертый тип инструментов составляют **инструменты контроля за версиями** проекта (наилучшей единицей контроля может служить отдельный модуль).
5) Пятый тип инструментов - это **библиотекари классов**, они позволяют сортировать классы и модули по определенному принципу (признаку), добавлять в библиотеку полезные классы по мере их создания и находить нужные для повторного использования.

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.

CASE-технология связана с популярными сегодня методами **визуального программирования** или **методами быстрой разработки приложений RAD** (Repid Application Development). Оба подхода имеют ряд общих черт, таких как применение для построения программ графических интерфейсов, объектных методов, современных способов тестирования (верификации).

То есть 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. Структура ЖЦ, основные фазы

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

Структура ЖЦ ПО по стандарту ISO/IEC 12207 базируется на трех группах процессов:

1) **Основные процессы**:
   a. разработка;
   b. использование (***сопровождение*** – устранение ошибок);
   c. эволюция (***модификация*** – внесение изменений в систему в ответ на изменившиеся требования).
2) **Вспомогательные процессы**: направлены на поддержку основных процессов ЖЦ ПО
   a. документирование;
   b. обеспечение качества ПО;
   c. верификация;
   d. аттестация;
   e. оценка;
   f. управление конфигурацией.
3) **Организационные процессы**: связаны с вопросами планирования и организации работ; создания коллектива разработчиков; контроля за сроком и качеством выполняемых работ; выбор CASE-инструментов
   a. управление проектом;
   b. улучшение самого ЖЦ;
   c. обучение и другие

Программный прототип - это ранняя реализация системы, в которой демонстрируется только часть ее функциональных возможностей.

Масштаб проекта и связанный с ним риск (вероятность того, что реализация функции окажет негативное влияние на график и бюджет). Масштаб проекта определяется следующими тремя переменными:

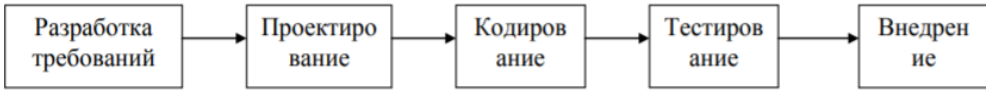
1) Набор функций, необходимых пользователю
2) Ресурсы, которыми располагает проект (труд разработчиков).
3) Время, выделенное на реализацию

13. Классические модели процесса разработки ПС

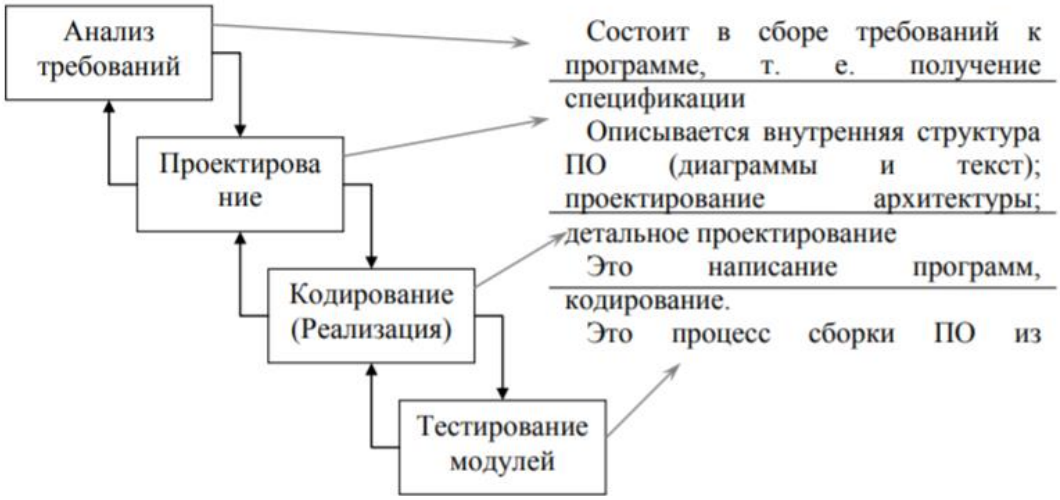
Любая модель процесса разработки ПО должна определять **кто** (какой член команды), **что делает** (какие действия), а также **когда** (действия по отношению к другим действиям) и **как** (детали и этапы действий) делает для достижения **цели** (цель – создание качественного продукта).

Каскадная модель *(Уинстон Ройс)*

До этого использовалась пошаговая модель:



Схематично каскадную модель можно изобразить так:

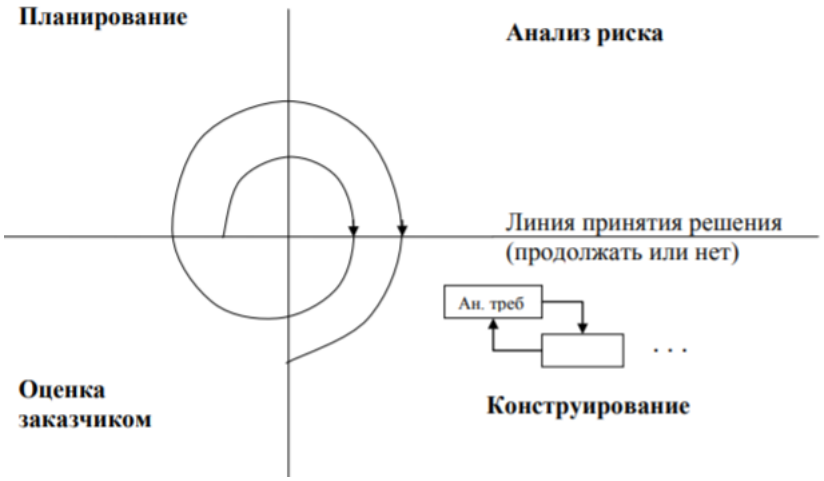


В модели “водопада” содержатся следующие усовершенствования строго “пошаговой” модели:

1. Фазы в модели 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: есть возможность вернуться на этап выше, если необходимо.
3. Ройсом предлагается параллельно с анализом требований и проектированием разработать систему – прототип.
4. Возросла роль анализа требований.
5. Каждый этап завершается выпуском полного комплекта документации.

Спиральная модель *(Боэм)*

Добавляется новый элемент – анализ риска. Схематично можно изобразить так:



1) **Планирование** – определение целей, требований, ограничений, составление плана разработки витка спирали (начиная со второго витка – на основе оценки и рекомендаций заказчика)
2) **Анализ риска**. (В начале на основе начальных требований, на следующих – на основе реакции заказчика)
3) **Конструирование** – разработка ПС. (Начальный макет -> Улучшенный макет -> … -> Готовая система)
4) **Оценивание** заказчиком текущих результатов конструирования.

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

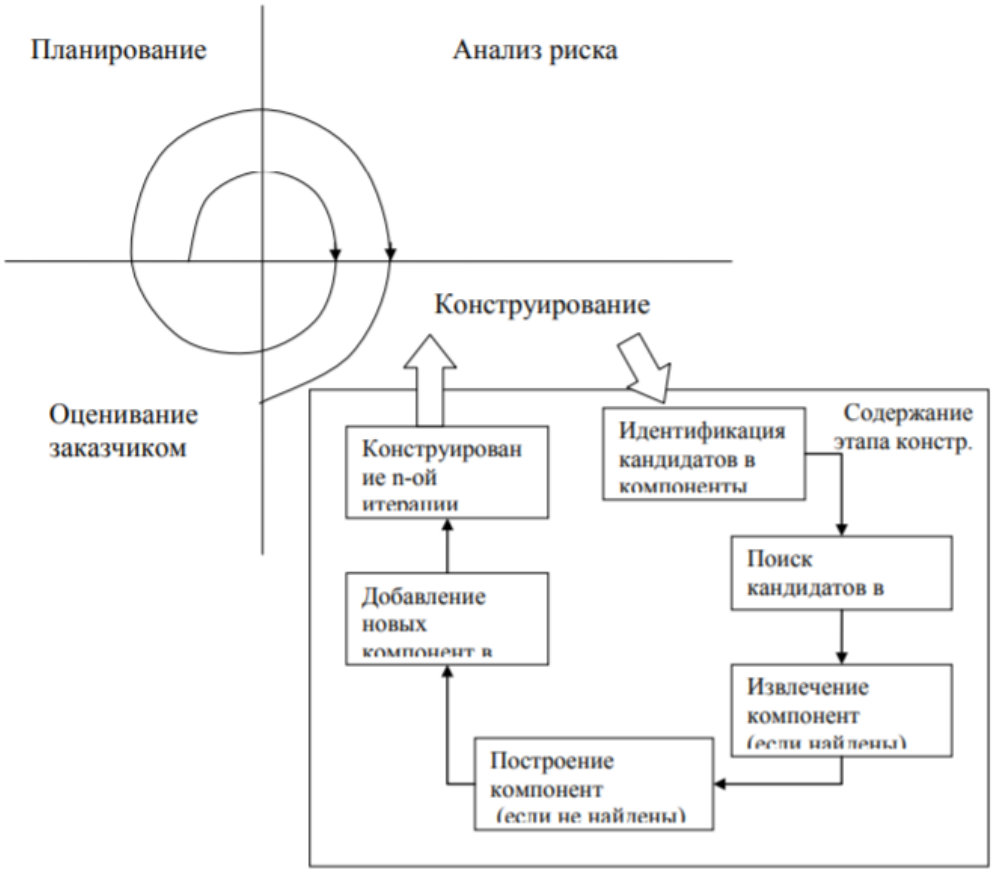
Достоинства спиральной модели:
1) наиболее реально (в виде эволюции) отображает разработку ПО;
2) позволяет явно учитывать риск на каждом витке эволюции разработки;
3) использует моделирование для уменьшения риска.

Недостатки спиральной модели:
1) повышенные требования к заказчику;
2) трудности контроля и управления временем разработки.

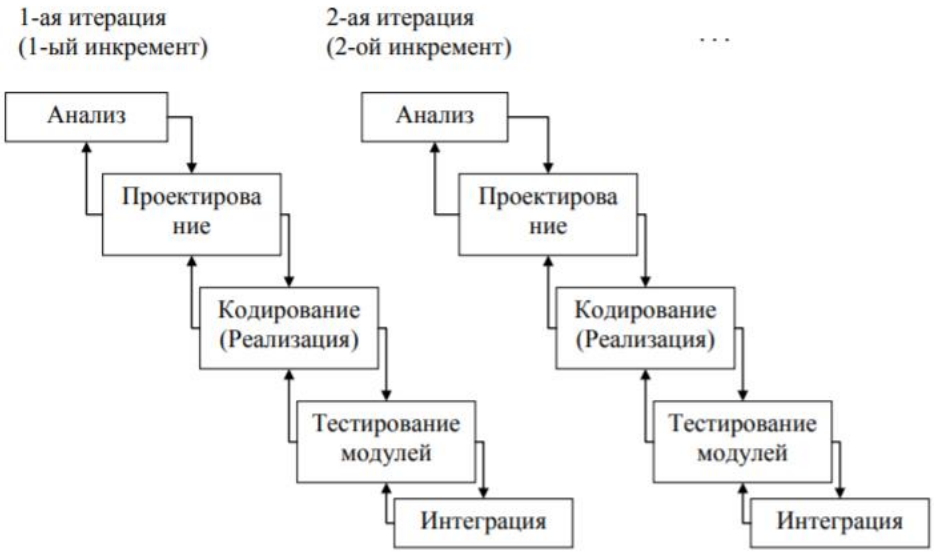
14. Модели процесса разработки ПС (компонентно-ориентированная, инкрементная, RAD-модель)

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

Схематичсно модель можно изобразить так:



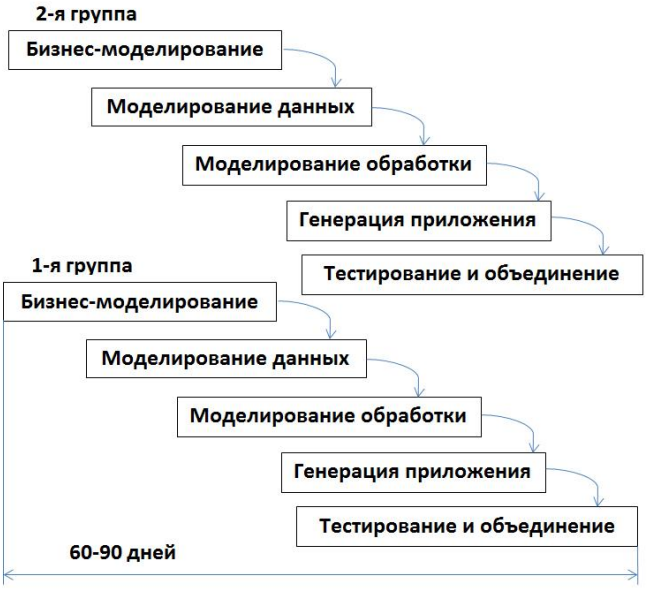
Инкрементная модель: в начале процесса конструирования определяются почти все пользовательские и системные требования. Затем конструирование выполняется в ввиде последовательности итераций, причем каждая итерация основывается на функциональных возможностях предыдущей.



RAD (Rapid Application Development) – это ЖЦ процесса проектирования, созданный для достижения более высоких 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 моделированияи разработки. Технология RAD предусматривает активное привлечение заказчика уже на ранних стадиях.

Принципы RAD технологии направлены на обеспечение трех основных ее преимуществ: *высокой скорости разработки*, *низкой стоимости* и *высокого качества*. 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, 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 характиристика **тяжеловесной методологии** (UP, RUP2, CMMI) – весь объем продстоящих 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, Kanban, Lean) используются при:
1) Частых изменениях требований
2) Малочисленной группе высококвалифицированных разработчиков
3) Грамотном заказчике, который согласен устовать в процессе разработки

Методология из **средней** категории (ICONIX, TSP, MSF) предствляет собой нечто среднее между тяжелыми и гибкими методологиями. Основная характиристика: 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 (класические или ближе к тяжелым)
4) Аутсорсинг (предпосылки к утяжелению проекта)

16. Унифицированный процесс разработки ПС. Модель процесса RUP

Это обобщенный каркас процесса, который может быть специализирован для широкого круга ПС, различных областей применения, уровней компетенции и размеров проекта. Основные характеристики (ниже подробнее):

1) управляется вариантами использования. Программная система создается для обслуживания ее пользователей. **Вариант использования** — это часть функциональности системы, необходимая для получения пользователем значимого для него, ощутимого и измеримого результата.
2) является **архитектурно-ориентированным**. Таким образом, архитектор в процессе работы совершает следующие шаги:
    a. Создает грубый набросок архитектуры.
    b. Работа с подмножеством выделенных вариантов использования.
    c. Созданная архитектура - база для полной разработки других вариантов использования.
3) является **итеративным** (итерации управляемы) и **инкрементным**.

Жизненный цикл унифицированного процесса.

Жизненный цикл UP разбивается на циклы, каждый из которых завершается поставкой выпуска 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) деятельности -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).

a. **актант** (actor) – некто (или нечто) вне системы, взаимодействующее с системой.

4) **модель –** абстракция, описывающая систему с определенной точки зрения и на определенном уровне абстрагирования.

5) **вариант использования** (use case, прецедент) – определение набора последовательностей действий, включая варианты, при которых система приносит отдельному актанту полезный и понятный результат.

6) **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.

17. XP-процесс (экстремальное программирование)

Экстремальное программирование (*Кент Бэк*) — облегченная (гибкая) методология. XP-процесс ориентированна 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.

Четыре базовых действия в XP-цикле:

1) кодирование,
2) тестирование,
3) выслушивание заказчика
4) 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.

Большинство принципов, поддерживаемых в XP (**минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, оптимальные стандарты кодирования** и т. д.), продиктованы здравым смыслом и применяются в любом упорядоченном процессе. Просто в XP эти принципы, как показано в таблице, достигают «экстремальных значений».

Базис XP образуют перечисленные ниже 12 методов.

1) **Игра планирования** (Planning game) — быстрое определение области действия следующей реализации путем объединения деловых приоритетов и технических оценок.
2) **Частая смена версий** (Small releases) — быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.
3) **Метафора** (Metaphor) — вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система.
4) **Простое проектирование** (Simple design)
5) **Тестирование** (Testing)
6) **Реорганизация** (Refactoring) — система реструктурируется, но ее поведение не изменяется; цель — устранить дублирование, улучшить взаимодействие, упростить систему или добавить в нее гибкость.
7) **Парное программирование** (Pair programming) — весь код пишется двумя программистами, работающими на одном компьютере.

8) **Коллективное владение кодом** (Collective ownership) — любой разработчик может улучшать любой код системы в любое время.
9) **Непрерывная интеграция** (Continuous integration) — система интегрируется и строится много раз в день, по мере завершения каждой задачи.
10) **40-часовая неделя** (40-hourweek) — как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счет сверхурочных работ.
11) **Локальный заказчик** (On-sitecustomer) — в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.
12) **Стандарты кодирования** (Coding standards) — должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях ПС.

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. Scrum – одна из самых популярных методологий гибкой разработки.

Основа Scrum — итеративная разработка. 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.

Каждую итерацию можно описать так: планируем — фиксируем — реализуем — анализируем. За счет фиксирования требований на время одной итерации и изменения длины итерации можно управлять балансом между гибкостью и планируемостью разработки.

Формальная часть Scrum состоит из трех ролей, трех практик и трех основных документов.

1) Роли
    a. **Product Owner**. Человек, отвечающий за разработку продукта. Обычно владелец продукта является представителем или доверенным лицом заказчика, а для компаний, выпускающих коробочные продукты, он представляет рынок, на котором реализуется продукт. Владелец продукта должен составить бизнес план, показывающий ожидаемую доходность и план развития с требованиями, отсортированными по коэффициенту окупаемости инвестиций. Исходя из имеющейся информации, владелец продукта подготавливает список требований, отсортированный по значимости. Владелец продукта – это единая точка принятия окончательных решений для команды в проекте, именно поэтому это всегда один человек, а не группа или комитет. Обязанности:
        i. Отвечает за формирование product vision
        ii. Управляет ожиданиями заказчиков
        iii. Координирует и приоритизирует Product Backlog
        iv. Предоставляет понятные и тестируемые требования к команде
        v. Взаимодействует с командой и заказчиком
        vi. Отвечает за приемку кода в конце каждой итерации
    b. **Scrum Master**. От Scrum Master-а во многом зависит самостоятельность, инициативность программистов, удовлетворенность сделанной работой,

атмосфера в команде и результат всей работы. Этот человек должен быть одним из членов команды разработки и участвовать в проекте как разработчик. Обязанности:

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

c. **Scrum Team** — группа, состоящая из 5–9 самостоятельных, инициативных программистов. Первая задача этой команды — поставить реально достижимую, прогнозируемую, интересную и значимую цель для итерации. Вторая задача — сделать все для того, чтобы эта цель была достигнута в отведенные сроки и с заявленным качеством. Цель итерации считается достигнутой только в том случае, если все поставленные задачи реализованы, весь код написан по определенным проектом «стандартам кодирования» (coding guidelines), программа протестирована полностью, а все найденные дефекты устранены. Обязанности команды таковы:

i. Оценка элементов Backlog
ii. Принятие решений по дизайну и имплементации
iii. Разработка софта и предоставление его заказчику
iv. Отслеживание собственного прогресса
v. Отчитывание перед Product Owner

2) Практики

a. **Sprint** - итерация длительностью 1 месяц. Результатом Sprint является готовый продукт (build), который можно передавать (deliver) заказчику. Подготовка к первой итерации, называемой спринт (Sprint), начинается после того, как владелец продукта разработал план проекта, определил требования и отсортировал их в количестве, достаточном для наполнения одной итерации. 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). Необходимо, чтобы все участники команды приняли на себя обязательство по реализации выбранной цели.

b. **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 каждому члену команды:

i. Что сделано вчера?

ii. Что будет сделано сегодня?
iii. С какими проблемами столкнулся?

Скрам Мастер собирает все открытые для обсуждения вопросы в виде Action Items, например в формате что/кто/когда

c. **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. Обсуждаются положительные и отрицательные способы совместной работы, анализируется, делаются выводы. Затем цикл замыкается, и начинается планирование следующего спринта

d. **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: производится его планирование и стартуются работы.

3) Артефакты

a. **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.

b. **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.

c. **Burndown Char**t. 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, учувствует в принятии решений;
4) Плотная работа с заказчиком, принимает непосредственное участие;

19. ICONIX-процесс

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 отдичает 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. С целью перехода от нечетких формулировок прецедента к очень точным и детальным диаграммам последовательности 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) ПС. Виды, достоинства, недостатки. Масштаб проекта и риски

ПРОГРАММНЫЙ ПРОТОТИП

Программный прототип (макет) - это ранняя реализация системы, в которой демонстрируют только часть ее функциональных возможностей. Прототипы необходимы для *снятия неопределенности в требованиях заказчика.*

Существует много способов разбиения прототипов на категории. Например, Дэвис выделяет следующие прототипы:

1) **Отбрасываемые**. Создаются, 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**. Воплощает срез функциональности приложения от интерфейса пользователя до сервисных функций.
5) **Горизонтальные**. Не реализуются все слои архитектуры, но воплощаются особенности интерфейса пользователя.
6) Интерфейсные. Много решений, чтобы строить в программах интерфейсы, уже не очень актуально.
7) Алгоритмические.

Выбор прототипа зависит от проблемы, которую надо решить. В любом случае, цель состоит в создании прототипа с наименьшими затратами. Если его создание обойдется слишком дорого, то следует сразу создавать реальную систему.

Достоинства прототипов:

1) Обеспечивают определение более полных требований к ПО.
2) Обеспечивают снятия 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).

Способы оценки масштаба проекта:

1) Упорядочивание списка требуемых функций, которые будут реализовываться в данной версии продукта, и задание их приоритетов (критический, полезный, важный).
2) Определение (приблизительное) объема трудозатрат, необходимых для каждой функции.
3) Оценка риска для каждой функции (*вероятности того, что реализация функции увеличит или время, или бюджет*)

Существуют два типа рисков.

1) **Устранимые**, которых можно избежать или которые можно предотвратить.
2) **Неустранимые**, которых невозможно избежать.

21. Содержание основных рабочих процессов по созданию ПО

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 и нефункциональными.

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*. Классификация видов тестирования:

1) **По объекту**:

a. Функциональное тестирование (functional testing)
b. Тестирование производительности (performance testing)
c. Нагрузочное тестирование (load testing)
d. Стресс-тестирование (stress testing)
e. Тестирование стабильности (stability/endurance/soak testing)
f. Юзабилити-тестирование (usability testing)
g. Тестирование интерфейса пользователя (UI testing)
h. Тестирование безопасности (security testing)
i. Тестирование локализации (localization testing)
j. Тестирование совместимости (compatibility testing)

2) **По знанию системы**:

a. Тестирование чёрного ящика (blackbox)
b. Тестирование белого ящика (whitebox)
c. Тестирование серого ящика (greybox)

3) **По степени автоматизации**:

a. Ручное тестирование (manual testing)
b. Автоматизированное тестирование (automated testing)
c. Полуавтоматизированное тестирование (semiautomated testing)

4) **По степени изолированности компонентов**:

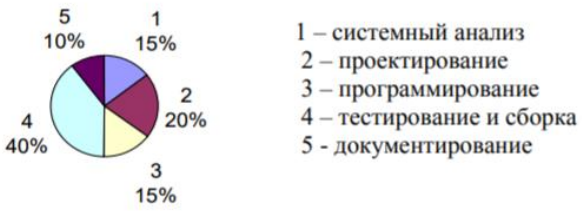
a. Компонентное (модульное) тестирование (component/unittesting)
b. Интеграционное тестирование (integration testing)
c. Системное тестирование (system/end-to-end testing)

5) **По времени проведения тестирования**:

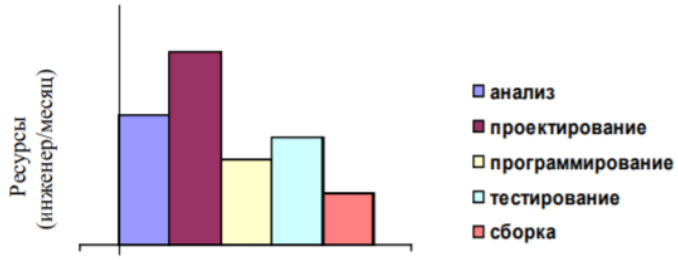
a. Альфа-тестирование(alpha testing)
b. Тестирование при приёмке (smoketesting)
c. Тестирование новой функциональности (new feature testing)
d. Регрессионное тестирование (regression testing)
e. Тестирование при сдаче (acceptance testing)
f. Бета-тестирование (beta testing)

22. Органиационные процессы (распределение ресурсов, управление проектом, способы организации коллектива разработчиков). Роли разработчиков

Приблизительные временные затраты на реализацию этапов разработки ПО:



При объектно-ориентированном подходе:



Главной проблемой и задачей руководства является *поддержание целостности основной идеи* в ходе работ, а также *контроль за качеством* продукта. Одним из методов, который используется в этих целях - это ***сквозной контроль*** или структурный контроль. Данный метод представляет собой серию проверок, проводимых на разных этапах цикла разработки, и состоит в регулярных встречах разработчиков ПО.

Контроль осуществляется посредством специальных ***контрольных сессий***. В них обычно участвуют от трех до шести человек, Обнаруженные ошибки соответствующим образом документируются (контрольный лист).

Способы организации коллектива разработчиков:

1) **Метод** "**монгольской орды**": вначале над проектом работает небольшая группа людей, а затем по мере необходимости подключают все большее число исполнителей.

2) **Способ "бригады главного программиста**": главному программисту придается группа специалистов. Ядро бригады составляют 3 человека: главный программист, его помощник и библиотекарь. *Главный программист* – это опытный специалист высокой квалификации, полностью отвечает за разработку проекта. П*омощник главного программиста* – тоже опытный и квалифицированный программист, в случае необходимости он всегда может заменить главного программиста. *Программист-библиотекар*ь хранит все записи проекта обеспечения разработки. Он осуществляет корректировку программ и их прогон, ведет библиотеку листингов, исходных и объектных модулей.

3) **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.

4) "**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:

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

23. Документирование программного продукта

Документация может содержать:

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

Продукты традиционных методов проектирования (SD, DD) — это диаграммы разбиения ПО на модули, схемы модулей, диаграммы процессов. При использовании стандартных методов разработки (SD и DD) в документацию должны входить: описание данных для всей системы (подсистем), а также описание входных и выходных данных для каждого модуля.

Продукты OOD - это диаграммы классов, модулей, процессов и объектные диаграммы. Так как эти результаты должны рассматриваться в общем контексте, то в документации они должны быть сгруппированы определенным образом. Проект каждого значительного сегмента системы (подсистемы) должен быть описан в отдельном документе. Этот документ должен отражать логическую схему системы и, следовательно, иметь, скорее всего, 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 данного документа может быть написано с помощью полуавтоматических методов. Документация, имеющая указанные три раздела, обычно носит название "**Руководство программиста**". Документация для пользователей - "**Руководство пользователю**". Менеджерам – «**Общее описание**». Системникам - "**Руководство системного программиста**".

В комплекс необходимой документации входят обычно такие документы:

1) *Техническое описание* (назначение, технические характеристики, методология).
2) *Справочное руководство* (управление, команды, сообщения об ошибках и т.д.)
3) *Рекламный буклет* (краткое описание назначения, наиболее важные технические характеристики, описание отличий от других аналогичных систем и т.д.)
4) *Руководство пользователя* (вся необходимая для эксплуатации информация)

Очень важной характеристикой программного продукта является **система обучения пользователей** (групповое, индивидуальное и автоматизированное).

24. Методы и средства структурного анализа

Анализ требований разрабатываемой системы - важнейший среди всех этапов ЖЦ. На этом этапе необходимо понять, что нужно сделать, и задокументировать это.

Проблемы: язык спецификации системы должен быть понятен и заказчику, и проектировщику, и программисту. Эта проблема может быть с помощью **методологии структурного анализа**.

Структурным анализом принято называть 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.
2) может присутствовать ограниченный контекст, который включает только существенные на каждом уровне детали.
3) использование строгих формальных правил записи.

В качестве базовых принципов используются следующие два: "**разделяй и властвуй**", **принцип иерархического упорядочивания**. Согласно этим двум принципам система может быть понята и построена по уровням, каждый из которых добавляет новые детали. Принципы разработки ПО:

1) Принцип *абстрагирования*
2) Принцип *сокрытия*
3) Принцип *концептуальной общности*
4) Принцип *полноты*
5) Принцип *непротиворечивости*
6) Принцип *логической независимости*
7) Принцип *независимости данных*
8) Принцип *структурирования*

Для целей моделирования систем используются три группы средств, которые иллюстрируют: функции системы; отношения между данными; поведение системы, которое зависит от времени.

В методах структурного анализа чаще всего применяются следующие средства (имеют графическую и текстовую нотации):

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. 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**. Кандидатами в классы и объекты следующее: люди, места, предметы, организации, концепции, события.
3) **Подход Йордана**. Кандидаты на классы и объекты: структуры, 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 в общем-то выполнять обязательства родительского класса, но он может иметь и свои услуги (дополнительные). После построения иерархий классов остается выявить **классы-клиенты** и **классы-поставщики** услуг и установить между ними соответствующие отношения (ассоциация, зависимость).

CRC-карточки (Component, Responsibility, Collaborator) - простой и эффективный способ анализа поведения будущей программной системы.

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

Анализ вариантов (Шлеер, Меллор). Предыдущие подходы сильно зависят от индивидуальных способностей и опыта аналитика. 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.

26. Методы объектно-ориентированного проектирования ПО

Модели объектно-ориентированный метод проектирования отражают иерархию классов и объектов.

Так, 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.

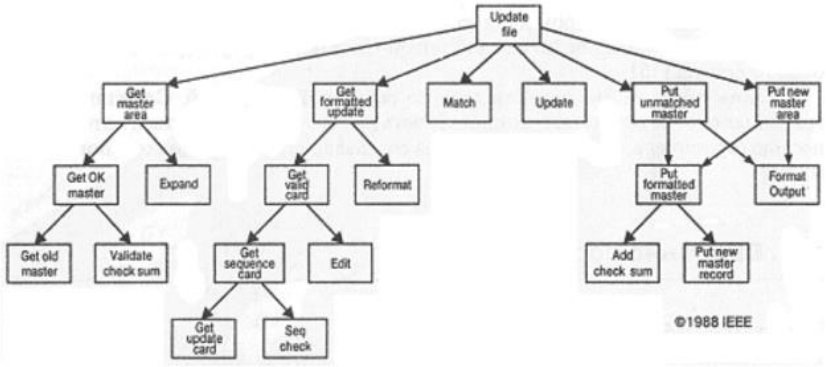
Иерархическое отношение классов подразумевает наследование - отношение обобщения в UML. Классы связанных между собой объектов связаны отношением ассоциации. Итак, ОО методология это: OOAnalysis + OODesign + OOProgramming.

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. Пример библиотечной системы

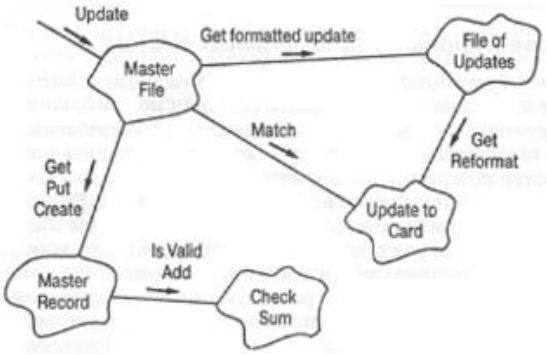
При проектировании сложной ПС необходимо разделять ее меньшие подсистемы, каждую из которых можно совершенствовать независимо. Декомпозиция вызвана сложностью программирования ПСоскольку именно эта сложность вынуждает делить пространство состояний ПС.

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/субъектами действия. Начать разделение системы надо либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения. Опыт показывает, что полезнее начинать с объектной декомпозиции для придания организованности ПС. 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. Виды абстракций (Сейдвиц и Старк):

1) *Абстракция сущности*. Объект представляет собой полезную модель некой сущности в предметной области.
    a. *Клиент* - объект, использующий ресурсы другого объекта;
    b. *Сервер* - объект, предоставляющий услуги клиенту.
    c. *Протокол* – отражает все возможные способы, которыми объект может действовать или подвергаться воздействию.
    d. *Инвариант* (*постусловия, предусловия*)- логическое условие, значение которого должно сохраняться
2) *Абстракция поведения*. Объект состоит из обобщенного множества операций.
3) *Абстракция виртуальной машины*. Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня.
4) *Произвольная абстракция*.

Поведение объекта характеризуется услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. Такой подход называют **контрактной моделью программирования**: внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство.

Инкапсуляция. Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется посредством скрытия информации и внутренней структуры объекта, реализация его методов. Практически это означает наличие двух частей в классе: интерфейса и реализации. Интерфейс отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя реализация описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Так, инкапсуляция - это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение.

Иерархия. Значительное упрощение сложных задач достигается за счет образования из абстракций иерархической структуры. 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.

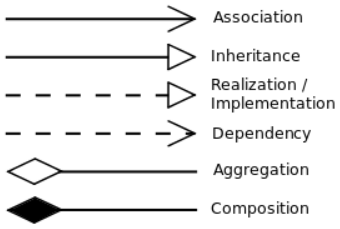
29. Унифицированный язык моделирования ПС (UML). Словарь, достоинства и возможности

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

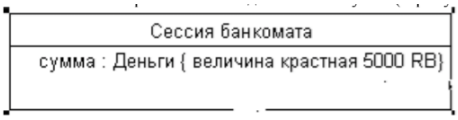
диаграмма может содержать любую комбинацию предметов и отношений, но напрактике ограничиваются только девятью видами диаграмм:

a. диаграммы прецедентов (вариантов использования);
b. диаграммы классов;
c. диаграммы объектов;
d. диаграммы последовательностей;
e. диаграммы сотрудничества (кооперации);
f. диаграммы состояний:
g. диаграммы деятельности;
h. диаграммы компонент;
i. диаграммы размещения;

30. Механизмы расширения в UML 2

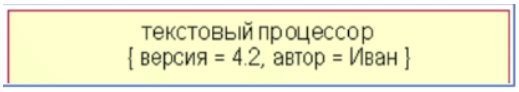
UML создавался изначально как открытый язык, который допускает расширения. Расширения позволяют отразить все нюансы, которые могут возникнуть при создании различных моделей. Механизмами расширения в UML являются: **ограничения**, **теговые величины** и **стереотипы**.

1) **Ограничение**(constraint) 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, например на 2

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** (stereotype) расширяет словарь языка, т.е. позволяет создавать новые виды строительных блоков, которые являются производными от существующих, но учитывают специфику проблемы. У стереотипа может быть другое визуальное представление, и они могут иначе обрабатываться при генерации кода. Изображается стереотип в двойных угловых скобках <<>>. Стереотип может быть задан на любом предмете в модели, а также на отношении. Таким образом, механизмы расширения позволяют адаптировать UML под новые технологии, под нужды конкретных проектов.

31. Диаграммы классов (точки зрения)

Одна из основных диаграмм языка UML – диаграмма классов. Она описывает классы (интерфейсы) и отражает отношения, существующие между ними. У диаграммы классов два главных элемента: классы (интерфейсы) и отношения между ними. Класс в UML служит для обозначения множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов. На значках класса обычно указывают важные в зависимости от точки зрения на диаграмму атрибуты и операции класса. В качестве имени класса принято употреблять существительное единственного числа. Имя класса может быть составным впереди добавлено имя пакета, куда входит класс.

Множественность. Иногда бывает необходимо ограничить количество экземпляров класса - по умолчанию на количество объектов нет ограничений. Множественностью класса называется количество его экземпляров. Она записывается в правом верхнем углу значка класса. Можно указать одно из следующих значений:

1) 1 – задать один экземпляр (класс singleton);
2) n–задать конкретное количество экземпляров;
3) * – задать любое количество экземпляров;
4) 0 - задать ноль экземпляров (в этом случае класс превращается в утилиту,которая предлагает свои свойства и операции).

Множественность применима не только к классам, но и к атрибутам (задается выражением в квадратных скобках после имени атрибута.

В диаграмме классов могут быть не только классы, но и пакеты, интерфейсы (логические сущности), файлы, узлы и Web-страницы. Подобные единицы на диаграммах классов изображают по-особому. Если в системе много классов, то на логическом уровне их группируют в пакеты (контейнеры). Распределение классов по пакетам дает возможность проектировщику подняться на более высокий логический уровень восприятия модели. Раскрывая содержимое пакета, переходим к отдельным элементам модели (чаще всего к классам).

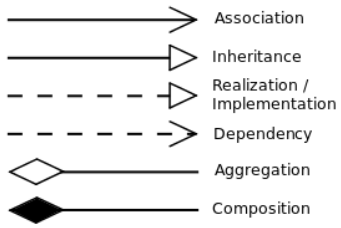
Точки зрения на диаграммы классов:

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):

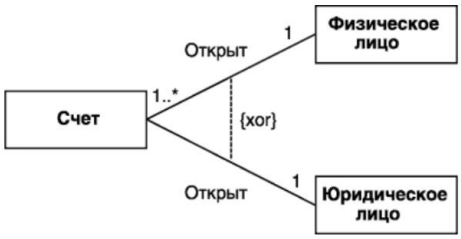
1) **Ассоциация** (равноправные классы Человек - Компания). Это отношение отображает структурные отношения между объектами классов. Это означает, что между объектами классов есть соединение, и через это соединение они могут посылать сообщения друг другу. Любая ассоциация обладает двумя ролями. Может показывать, сколько экземпляров одного класса взаимодействуют друг с другом. Могут быть двунаправленные и однонаправленные.
2) **Обобщение** (наследование).
3) **Зависимость**. Показывает, что один класс ссылается на другой, и изменения в первом повлекут изменения во втором. Стрелка от того, кто зависит к тому, от кого.
4) **Реализация**. (Обработчик каталога - каталог) Это семантическое отношение между классами, в котором класс-приемник выполняет реализации операций интерфейса класса-источника. Стрелка к тому, кто реализует.
5) **Агрегация** (включение по ссылке Трупа Актер). Задает более сильную форму однонаправленной ассоциации, которая устанавливает соотношение частей и целого.
6) **Композиция** (физическое включение). Показывает физическое вхождение части в целое: удаляется целое – удаляются и все его части.



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

Ассоциация (association) - семантическое отношение между двумя и более классами, которое специфицирует характер связи между соответствующими экземплярами этих классов.

Бинарная. Каждому студенту может соответствовать только одна группа, мощность связи 1 к 1. Может быть ненаправленной (симметричной) или направленной. Частный случай бинарной ассоциации - ***рефлексивная ассоциация***, которая связывает класс с самим собой. Если направленная, то направление стрелки указывает на то, какой класс является первым, а какой – вторым. Еще один частный случай ассоциации – ***исключающая ассоциация***. Семантика данной ассоциации указывает на то, что из нескольких потенциально возможных вариантов данной ассоциации в каждый момент времени может использоваться только один.



N-арная ассоциация. Каждому студенту может соответствовать несколько предметов, мощность связи один-ко-многим. Такая ассоциация связывает отношением более чем три класса, при этом класс может участвовать в ассоциации более чем один раз. Каждый экземпляр n-арной ассоциации представляет собой n-арный кортеж, состоящий из объектов соответствующих классов. В этом контексте бинарная ассоциация является частным случаем n-арной ассоциации, когда значение n=2, но имеет собственное обозначение. Графически n-арная ассоциация обозначается ромбом, от которого ведут линии к символам классов данной ассоциации.

Класс ассоциация (association class) - модельный элемент, который одновременно является ассоциацией и классом. С этой целью в языке UML вводится в рассмотрение специальный элемент - концевая точка ассоциации или **конец ассоциации** (AssociationEnd), который графически соответствует точке соединения линии ассоциации с отдельным классом.

К дополнительным обозначениям относится имя роли отдельного класса, входящего в ассоциацию. **Роль** (role) - имеющее имя специфическое поведение некоторой сущности,

рассматриваемой в определенном контексте. Роль может быть статической или динамической.

34. Отношение зависимости в диаграммах классов этапа проектирования. Стеротипы 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. При генерации кода для классов с отношением зависимости инструмент, как правило, не добавляет к ним никаких атрибутов, но создает дополнительные операторы, специфичные для языка программирования.

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.

Еще можно заметить, что отношение зависимости самое разнообразное среди других типов отношений. В нотации UML предусмотрено 17 различных зависимостей (и не только для классов). Для их различия задается стереотип, например, bind (подстановка параметров в шаблон), call (зависимость между операциями), instanceOf, derive («может быть вычислен по»), instantiate.

35. Диаграммы вариантов использования, реализация 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. В состав диаграммы UseCase входят следующие элементы:

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 бытьлюди, внешняя система, время
3) **отношения** между ними (связи).

Главное в диаграммах UseCase – они не зависят от реализации. Каждый вариант использования должен представлять собой завершенную транзакцию между пользователем и системой.

Рассмотрим ***отношения (связи) в диаграммах UseCase***. Между актером и прецедентом возможен только один тип отношения – **ассоциация**. Как любая ассоциация она может быть помечена именем, ролью, мощностью. Направление стрелки показывает, кто инициирует связь. Между актерами допустимо отношение обобщения. Между вариантами использования определены отношения ***обобщения***

*(*UseCase-потомок наследует поведение родителя и, кроме того, может дополнить или переопределить поведение UseCase-родителя*)* и две разновидности отношения зависимости – ***включение*** *(*позволяет одному UseCasesзадействовать функциональность другого*)* и ***расширение*** *(*базовый элемент неявно включает поведение другого в точке, которая должна быть определена расширяющим элементом UseCase*).*

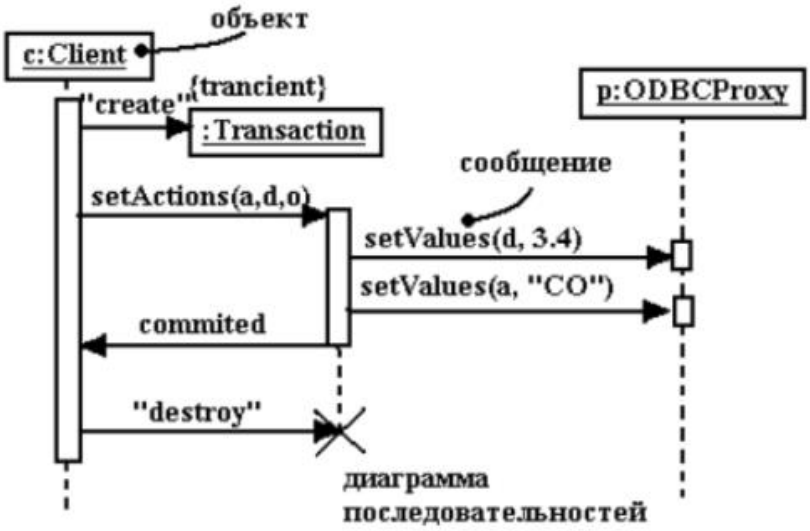
Как правило, все отклонения от нормального процесса помещаются в отдельный вариант использования, и затем между конкретным (нормальным) вариантом использования и вариантом, который обрабатывает отклонение (абстрактный вариант), проводится связь типа расширения.

Таким способом можно отделить обязательное поведение от необязательного, которое не всегда исполняется, а только при определенных условиях. Этот процесс иначе называется расщеплением варианта использования. Процедура расщепления элемента UseCase обычно проводится на этапе проектирования системы (возможно даже в процессе реализации программы).

Реализации вариантов использования. Реализации вариантов использования (или кооперации), как правило, фиксируются на Логическом уровне представления модели и имеют те же имена, что и варианты использования в UseCaseModel. Обозначения их также схожи, 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 языка UML, которые отражают взаимодействие объектов друг с другом, получили название **диаграмм взаимодействий**. 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

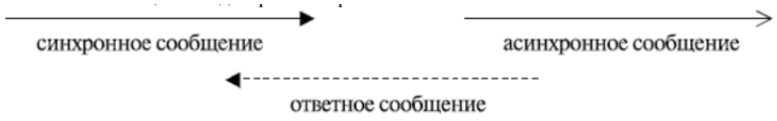
объектами отображается с помощью горизонтальных линий, проведенных между соответствующими вертикальными линиями.

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. Большая часть объектов, представленных на диаграмме взаимодействий, существует на протяжении всего взаимодействия, поэтому их изображают в верхней части диаграммы, а их линии жизни прорисованы сверху донизу. Объекты могут создаваться и во время взаимодействий. Линии жизни таких объектов начинаются с получения сообщения со стереотипом «*create*». Объекты могут также уничтожаться во время взаимодействий. В таком случае их линии жизни заканчиваются получением сообщения со стереотипом «*destroy*».

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. Верхняя грань прямоугольника выравнивается по временной оси с моментом начала действия, нижняя – с моментом его завершения. Вложенность фокуса управления, вызванную рекурсией (то есть обращением к собственной операции) или обратным вызовом со стороны другого объекта, можно показать, расположив другой фокус управления чуть правее своего родителя (допускается вложенность произвольной глубины).

Чаще всего с помощью диаграмм взаимодействий моделируют *последовательные неветвящиеся потоки управления*. Более сложные потоки, содержащие итерации и ветвления, удобней моделировать с помощью **диаграмм деятельности**. Хотя, если итераций и ветвлений немного, то их можно показать и на диаграммах взаимодействий с помощью *.

Формы управления потоком. Наиболее общую форму управления задает процедурный поток или поток синхронных сообщений. Процедурный поток рисуется стрелками с заполненными наконечниками ().



Работа с **синхронным сообщением** подчиняется следующему правилу: передатчик ждет до тех пор, пока получатель не примет и не обработает сообщение.

Хорошо структурированная диаграмма взаимодействий обладает следующими свойствами:

1) акцентирует внимание только на одном аспекте динамики системы;
2) содержит только такие варианты использования и актеры, которые важны для понимания этого аспекта;
3) содержит только такие детали, которые соответствуют данному уровню абстракции, и только те дополнения, которые необходимы для понимания системы;

37. Диаграммы состояний

На диаграмме состояний отображают ЖЦ одного объекта (или класса объектов), начиная с момента его создания и заканчивая разрушением. С помощью таких диаграмм удобно моделировать динамику поведения объекта класса. Например, объект Сотрудник может быть нанят, уволен, проходить испытательный срок, находиться в отпуске или в отставке. В каждом из этих состояний объект класса сотрудники может вести себя по-разному.

Диаграммы состояний не требуется создавать для каждого класса. Многие проекты вообще обходятся без них. Если динамика класса важна, то для него полезно создать диаграмму состояний. Такие классы обычно имеют много (≥4) различных состояний, то есть поведение объекта такого класса существенно, и есть смысл его смоделировать. Поведение таких классов управляется ***событиями***.

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. На значках некоторых состояний можно указать действия, ассоциированные с этими состояниями объекта (*входные* - поведение, которое всегда выполняется, когда объект переходит в данное состояние, *выходные* - осуществляется как составная часть процесса выхода из состояния и *деятельность* - поведение, которое реализует объект, находясь в данном состоянии).
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).

Начальное и конечное состояние На диаграмму состояний обычно добавляют два специальных состояния объекта – начальное и конечное. Начальное состояние изображается в виде закрашенного кружочка: от него проводится переход к первоначальному состоянию. Конечное состояние изображают в виде значка «бычий глаз».

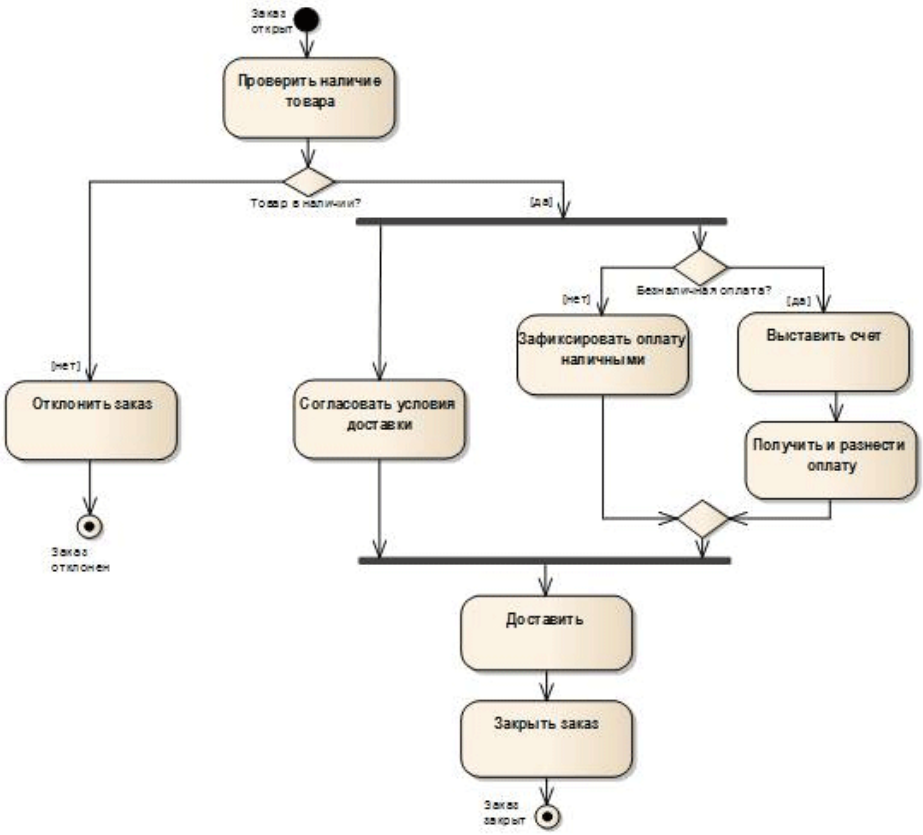
38. Диаграммы активности (деятельности)

Диаграммы деятельностей используются для описания поведения систем. Эти диаграммы особенно полезны в сочетании с потоками работ, а также в описании поведения, которое включает параллельные процессы.

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.



Наличие между диаграммой деятельностей и блок-схемой в том, что блок-схемы показывают последовательные процессы, а диаграммы деятельностей могут поддерживать дополнительно *параллельные процессы*. Можно графически изобразить все ветви и показать, когда их необходимо синхронизировать.

39. Каркасы и паттерны

Образец (**паттерн**) **проектирования** предлагает типичное решение типичной проблемы в определенном контексте. Это шаблоны взаимодействующих классов (объектов) и методов. Образцы используются как на уровне архитектуры, так и на уровне детального проектирования.

Все паттерны делят на 3 категории: *структурные* (Адаптер, Компоновщик, Декоратор, Мост, Фасад. Используются с целью образования более крупных структур из классов и объектов для получения новой функциональности. Когда система больше зависит от композиции объектов, чем от наследования классов), *порождающие* (Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Синглтон) и *паттерны поведения* (Итератор, Интерпретатор, Посредники, Наблюдатель. Позволяет следить за поведением объектов).

Паттерны можно рассматривать как крупные строительные блоки. Обозначение паттерна в нотации UML имеет следующий вид:

Каркас (**framework**) – это коллекция классов, используемых в нескольких различных приложениях. Это архитектурный образец, который предлагает расширяемый шаблон. Поэтому каркасные классы, которые используются в проекте, являются частью

архитектуры приложения. Каркас – это больше, чем механизм. 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).

Итак, каркасы описывают структуру и поведение системы в целом, а паттерны описывают структуру и поведение сообщества классов. Паттерны используются и в каркасах, и в процессе детального проектирования.

40. Понятия, определения теории тестирования. Подходы, стратегии, критерии

Тестирование ПО – это процесс анализа или эксплуатации ПО с целью выявления дефектов. Тестирование – это плановая и упорядоченная деятельность.

Дефект (или баг) – это несоответствие требованиям к программному продукту или его функциональной спецификации. Дефект может возникнуть из-за: некорректных требованиий, неверного толокания требований программистом, неправильно настроенного тестового оборудования.

Ожидаемый результат – предполагаемое корректное поведение ПС. 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.

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

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

Статическое тестирование(без прогона кода)/динамическое тестирование (с прогоном кода)

УРОВНИ ТЕСТИРОВАНИЯ

1) ***Модульное тестирование***. Позволяет проверить функционирование отдельно взятого элемента системы: функция, класс, компонента.
2) ***Интеграционное тестирование***. Процесс проверки взаимодействия между программными компонентами/модулями. Может делиться на юнит-тестирование (пограммисты) и интеграционное тестирование (тестировщики) приложения.
3) ***Системное тестирование***. Фокусируется оно на проверке, как функциональных требований, так и нефункциональных – требованиях безопасности, производительности, точности, надежности т.п. На этом уровне также тестируются интерфейсы к внешним приложениям, аппаратному обеспечению, ОС и т.д
4) ***Функциональное тестирование***. (Белый ящик)Подразделяется на ручное и автоматизированное.

СТРАТЕГИИ: «белого ящика», «черного ящика»

ВИДЫ ТЕСТИРОВАНИЯ

1) Инсталяционное
2) Регрессионное
3) Новой функциональности
4) Конфигурационное
5) На удобство эксплуатации

6) Интернационализации
7) Локализационное
8) Positive (что не делает, а должно) / Negative (что делает, чего не должно)
9) Исследовательское
10) Безопасности
11) Производительности
12) Нагрузочное
13) Стрессовое

41. Критерии тестирования стратегии «черного ящика»

Метод черного ящика (blackboxtesting), тесты которого разработаны исходя из знаний функциональных и бизнес 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. То есть он запускает приложение на выполнение и тестирует его функциональность, используя пользовательский интерфейс для ввода входных данных и получая выходные. Но как при этом обрабатываются входные данные, он не знает. Цель данного метода – проверить работу всех функций приложения на соответствие функциональным требованиям.

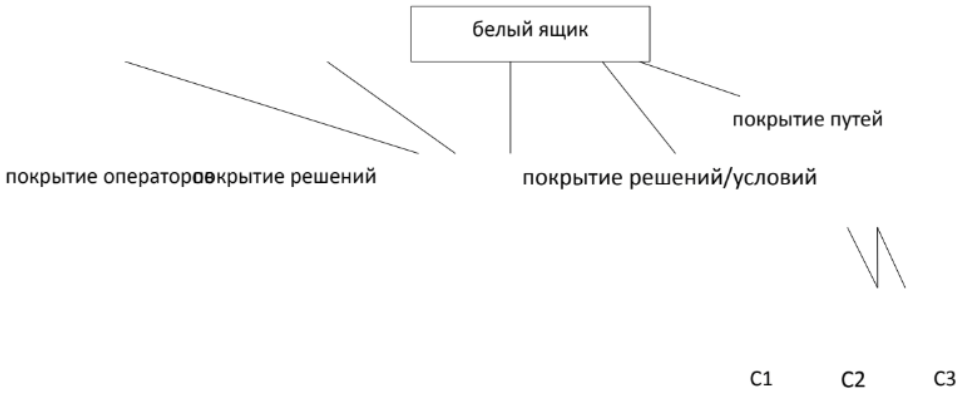
1. *Эквивалентное разбиение*. Осуществляется в 2 этапа:
    a. *Выделение классов эквивалентности*. Путем рассмотрения каждой входной области данных программы и разбиении её на ≥2 группы. Класс эквивалентности – набор данных с общими свойствами. Программы, обрабатывая разные элементы одного класса эквивалентности, должны вести себя одинаково. Два вида входных классов ~: *допустимые*, *недопустимые* данные.
    b. *Построение тестов*. Надо, чтобы все входные классы ~ были покрыты.
2. *Анализ граничных значений/условий*. Граничные условия – это ситуации, которые возникают непосредственно на границе классов ~, а также выше или ниже этих границ. Анализ граничных условий отличается от ~ разбиения следующими моментами:
    a. Выбор элемента в классе ~ осуществляется не произвольно, а так, чтобы проверить каждую границу класса
    b. Рассматриваются не только входные классы ~, но и правильные выходные классы ~
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»

Для тестирования программного кода без его непосредственного запуска применяется метод белого ящика. Этот метод еще называют методом прозрачного или стеклянного ящика.

Относится к структурному тестированию. Тесты основаны на знании кода приложения и его внутренних механизмов. Соответственно концепция структурного тестирования связана с тестированием внутренней структуры исходного кода ПО. Метод белого ящика обычно применяется на стадии, когда приложение еще не собрано воедино, но необходимо проверить каждый из его компонентов, модулей, процедур и подпрограмм. Следовательно,

структурное тестирование тесно взаимосвязано с компонентным или модульным тестированием (unit testing).



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 опраторах 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 одинраз. 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)

Для обнаружения ошибок в рамках модуля тестируются его важнейшие управляющие пути. Относительная сложность тестов и ошибок определяется как результат ограничений области тестирования модулей. Принцип тестирования — «белый ящик», шаг может выполняться для набора модулей параллельно. Тестированию подвергаются:

1) *интерфейс* модуля;
2) *внутренние структуры данных*; гарантирует целостность сохраняемых данных
3) *независимые пути*; гарантирует выполнение всех операторов модуля не менее одного раза, а также прогон всех реализуемых путей. При тестировании путей выполненияобнаруживаются следующие категории ошибок: ошибочные вычисления, некорректные сравнения,неправильный поток управления.
4) *пути обработки ошибок*; Для защиты от ошибочных условий
5) *граничные условия.*

Тестирование модулей обычно рассматривается как дополнение к этапу кодирования. Оно начинается после разработки текста модуля. Так как модуль не является автономной системой, то для реализации тестирования требуются дополнительные средства – какая-нибудь программная среда. Дополнительными средствами являются драйвер тестирования и заглушки. Драйвер — управляющая программа, которая принимает исходные данные и ожидаемые результаты тестовых вариантов, запускает в работу тестируемый модуль, получает из модуля реальные результаты и формирует донесения о тестировании.

44. Классический процесс тетстирования ПО (методика тестирования)

Классический процесс тестирования обеспечивает проверку результатов, полученных накаждом 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 компонентамит компьютерной системы.

Методика тестирования ПС может быть предоставлена в виде разворачивающей спирали. Рассмотрим кратко каждый шаг процесса тестирования :

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**. 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. Тестирование интеграции Тестирование правильности. Системное тестирование

ТЕСТИРОВАНИЕ ИНТЕГРАЦИИ

Берутся модули, протестированные на предыдущем шаге как отдельные элементы, и собираются в рабочую программу. Тесты проводятся для обнаружения ошибок интерфейса. Существует 2 способа тестирования интеграции:

1) **монолитный способ.** Каждый модуль тестируется по отдельности, а затем – все сразу соединяются в рабочую программу;
2) **пошаговый способ**. Каждый модуль пошагово подключается к набору ранее оттестированных модулей. Пошаговый способ тестирования интеграции может быть ***нисходящим*** и ***восходящим***. В нисходящем подходе методы объединяются движением сверху вниз по управляющей иерархии, начиная от главного управляющего модуля. Подчиненные модули добавляются в структуру или в результате поиска в глубину, или в результате поиска в ширину. *Достоинство* нисходящей интеграции: ошибки в главной, управляющей части системы выявляются в первую очередь. *Недостатоκ*: возникают трудности в ситуациях, когда для полного тестирования на верхних уровнях нужны результаты обработки с нижних уровней иерархии. Существует 3 способа борьбы с этим недостатком:

   a. откладывать некоторые тесты до замещения заглушек реальными модулями;
   b. разрабатывать заглушки, частично выполняющие функции модулей;
   c. подключать на некоторых путях модули движением снизу вверх.

При **восходящем тестировании** интеграции сборка и тестирование системы начинаются с модулей, которые располагаются на нижних уровнях иерархии. Для этого подхода нет необходимости в заглушках, зато нужны драйверы для модулей верхних уровней. *Достоинство* восходящего тестирования: упрощается разработка тестовых вариантов, отсутствуют заглушки. *Недостатоκ*: система не существует как объект до тех пор, пока не будет добавлен последний модуль.

Можно отметить, что возможен комбинированный подход: для верхних уровней - нисходящее тестирование, для нижних - восходящее тестирование. Метод восходящего тестирования полезен в тех случаях, когда программа состоит изотдельно компилируемых физических модулей.

ТЕСТИРОВАНИЕ ПРАВИЛЬНОСТИ (ФУНКЦИОНАЛЬНОЕ)

После окончания тестирования интеграции ПС собрана в единый корпус, интерфейсные ошибки обнаружены и откорректированы. Следующим шагом тестирования ПС является тестирование правильности. Цель — подтвердить, что функции, описанные в спецификации требований к ПС, соответствуют ожиданиям заказчика. Подтверждение правильности ПС выполняется с помощью тестов «черного ящика», демонстрирующих соответствие требованиям. При обнаружении отклонений от спецификации требований создается список недостатков.

Важным элементом подтверждения правильности является **проверка конфигурации** ПС. Минимальная конфигурация ПС включает следующие базовые элементы:

1) системную спецификация;
2) спецификацию требований к ПС;
3) спецификация проектирования;
4) план программного проекта;
5) предварительное руководство пользователя;
6) листинги исходных текстов программ;
7) план и методику тестирования;
8) тестовые варианты и полученные результаты;
9) руководства по работе и инсталляции;
10) описание базы данных;
11) руководство пользователя по настройке;
12) документы сопровождения; отчеты о проблемах ПС; запросы сопровождения; отчеты оконструкторских изменениях;
13) стандарты и методики конструирования ПС.

Проверка конфигурации гарантирует, что все элементы конфигурации ПС правильно разработаны, учтены и достаточно детализированы для поддержки этапа сопровождения в жизненном цикле ПС. Для обнаружения ошибок, которые способен найти только конечный пользователь, используют процесс, включающий **альфа**- и **бета**-тестирование. Альфа-тестирование проводится заказчиком в организации разработчика. Разработчик фиксирует все выявленные заказчиком ошибки и проблемы использования. Бета-тестирование проводится конечным пользователем в организации заказчика.

СИСТЕМНОЕ ТЕСТИРОВАНИЕ

Системное тестирование подразумевает выход за рамки области действия ПО и проводится не только программным разработчиком.

1) Тестирование восстановления. Многие ПС должны восстанавливаться после отказов и продолжать обработку в пределах заданного времени. В некоторых случаях система должна быть отказоустойчивой.
2) Тестирование безопасности (в основном для банковских систем).
3) Стрессовые тесты проектируются для навязывания программе ненормальных ситуаций, называется "расшатать" систему.

46. Особенности тестирования ООП программ

Возможности ОО ЯП оказывают влияние на некоторые аспекты тестирования: наследование классов, полиморфизм, интерфейсы, инкапсуляция (некоторые операции должны быть включены в интерфейс класса специально для целей тестирования.

Изменение процесса разработки ПО (смещение акцентов на анализ и проектирование) также оказывают очевидное воздействие на тестирование. Здесь все еще применяется тестирование программных модулей, хотя понятие модуля теперь изменилось. Вместо тестирования модулей в процедурной программе теперь тестируются реализованные методы классов и независимые подпрограммы).

По-прежнему осуществляется проверка функционирования и взаимодействия компонентов системы, регрессионное тестирование. Основные отличия процесса тестирования объектных программ от традиционных (процедурных) следуют из самого подхода к разработке ПО. А именно: **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. Это означает, что процесс тестирования может переплетаться с процессом разработки.

Многократно повторяется процесс:

1) Немного анализа
2) Немного проектных решений
3) Немного программных кодов
4) Тестирование.

Очень хорошие результаты показывает тестирование, выполняемое в режиме многократного повторения. Системное тестирование и приемочные испытания следуют за последней тестирующей итерацией. Тем не менее, если ПС можно разрабатывать путем последовательно наращивания ее функциональных возможностей (итеративный и инкрементный подход), то системное тестирование можно проводить после каждого такого наращивания.

Рассмотрим кратко, какие виды тестирования должны проводиться для ОО ПО. На практике следуют такому порядку:

1) Тестирование моделей (аналитических и проектных).
2) Тестирование классов (вместо тестирования модулей).
3) Тестирование взаимодействий и функционирования компонентов.
4) Тестирование подсистем и систем.
    a. Приемочные испытания. Основной метод – метод целенаправленной проверки (инспекции). Для этой цели используются следующие виды диаграмм: диаграммы UseCase; диаграммы последовательностей, классов анализа и проектные диаграммы; диаграммы состояний объектов класса и действий.

47. Тестирование классов

Тестирование классов означает проверку реализации класса на соответствие его спецификации. В первом приближении тестирование классов аналогично тестированию модулей в традиционных процессах тестирования. А именно – разрабатывается тестовый драйвер, который создает один или большее число экземпляров класса. Драйвер будет осуществлять прогон тестов. 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:

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

Самый простой критерий тестирования класса – это полное покрытие его программного кода, но чаще всего тестируемый класс рассматривается как «черный ящик», т.е. тесты готовятся на основе спецификации класса. В любом случае 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.

Разработка тестовых драйверов

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 обычно входит удаление любого созданного им экземпляра. На практике широко используются 3 критерия построения тестов для классов:

1) покрытие, ориентированное на состояния;
2) покрытие, ориентированное на ограничения; все пред и постусловия
3) покрытие программного кода.

48. Тестирование взаимодействия классов и функционирования компонентов

У большинства классов имеются партнеры по сотрудничеству, т. е. методы такого класса взаимодействуют с экземплярами других классов. Рассмотрим случаи, когда между классами установлены отношения ассоциации.

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. Процесс тестирования взаимодействий классов заметно облегчается, если число связей между ними не велико.

Тестирование иерархий классов.

Наследование в ОО методологии представляет собой мощный механизм, который обеспечивает многократное использование интерфейсов и реализаций классов. На стадиях анализа и проектирования отношение наследования может быть выражено одним из двух способов:

1) **Как специализация** некоторого класса, который уже был определен (интерфейсы, как правило, совпадают, а подкласс реализует некоторую операцию, заявленную в родительском классе).

2) **Как обобщение** одного или большего числа классов, которые уже были определены (подкласс расширяет функции родительского класса). Сильной стороной такой технологии является то, что программный код, который обеспечивает тестирование классов иерархии, допускает многократное использование. Таким образом, тестовые драйверы для подклассов могу быть получены из тестовых драйверов для базового класса. Реализация классов существенно упрощается, если она выполняется по иерархии сверху вниз.

49. Тестирование подсистем и систем. Приемочные испытания

Если определены спецификации подсистем и систем (должны быть отражены прежде всего в диаграммах UseCase и взаимодействий), 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

Такие испытания устраиваются заказчиком 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.

50. Основы автоматизированного тестирования

Для тестирования ПО, как и для любого рода деятельности, смысл автоматизации в повышении эффективности работы. Преимущества от внедрения автоматизации в целом следующие:

1) Скорость выполнения по сравнению с ручным тестированием значительно выше
2) Есть возможность запуска тестов в нерабочее время
3) Возможность повтора тестов, причем в точности так, так как было задумано, и с теми же данными
4) Исключение ошибок
5) Накопление результатов в формализованном виде
6) Освобождение тестировщиков от регулярного выполнения одних и тех же тестов,
7) Управление несколькими машинами одновременно - распределенный тест
8) Взаимодействие с базами данных "напрямую"
9) Перезагрузка машины в другую ОС и вход в систему под разными пользователями
10) Поиск "поломанных" ссылок на веб-страницах
11) Создание теста на одном браузере и его запуск на другом без изменений

Инструментальные программные средства для организации автоматизации могут:

1) Понимать и распознавать объекты
2) Объектно-ориентированные языки программирования
3) Могут вызывать DLL
4) Могут обращаться к коду приложения
5) Позволяют создавать кроссс-платформенные и кросс-браузерные тесты
6) Восстанавливаться после сбоев и продолжать тесты

При грамотном применении явных недостатков у автоматизации не так уж и много. Во-первых, регрессионные тесты, как правило, новых багов не находят - только старые, ужеисправленные проблемы, если они снова "вылазят". Это хорошо, но хотелось бы большего. А чтобы тест находил новые проблемы, он должен уметь или генерировать различные тестовые ситуации самостоятельно или чтобы он брал для тестирования различные данные.

Во-вторых, главный выигрыш от использования автоматического тестирования всегда приходится на фазы проекта, следующие за той, в которой мы начали его внедрение. Кроме всего прочего не стоит забывать, что разработка и отладка тестов требует больших затрат времени.

Наиболее выгодными тестами для автоматизации являются те, которые нужно будет проводить большое количество раз. Smoke test (приемочные тесты) - проводятся для каждого build приложения. Различные конфигурации - это могут быть как различные настройки самого тестируемого продукта, так и различные конфигурации тестовой среды (наборы ОС, веб-браузеров, других компонентов, типа JRE). Ну а еще другая самая очевидная цель - это автоматизация регрессионного тестирования

Оценка трудозатрат и целесообразности автоматизации

В большинстве случаев для оценки выгоды от автоматизации вполне достаточно простой арифметикии оценки только временных затрат. При оценке затрат (подразумевается что тесты для ручного тестирования уже разработаны и документированы) нужно учитывать следующие величины:

1) Время на проход теста вручную
2) Количество таких проходов
3) Время на разработку и отладку автоматического теста
4) Время его работы
5) Время на исправление скрипта при изменениях в приложении