ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

Б1.В.ДВ.09.02 Основы компьютерного проектирования и моделирования АСОИ

Направление подготовки *09.03.01 Информатика и вычислительная техника*

Профиль образовательной программы *Автоматизированные системы обработки информации и управления*

Форма обучения *очная*

1. КОНСПЕКТ ЛЕКЦИЙ

1.1 Лекция № 1, 2 (4 часа)

Тема: «Общие принципы проектирования информационных систем**»**

1.1.1 Вопросы лекции:

1. Объект проектирования.
2. Процесс проектирования.
3. Формальная логика.
4. Процесс образования понятия.
5. Диалектическая логика.
6. Содержательно-генетическая логика.
7. Логическая схема проектирования.
8. Логические отношения.
9. Определение задач проектирования.
10. Определение логической схемы проектирования (методологии проектирования).
11. Решение задач проектирования.
12. Понятие системного подхода и системного анализа.
13. Документ. Электронный документ. Информационная система.
14. Информационная технология.

1.1.2 Краткое содержание вопросов:

1. Объект проектирования.

Различные виды проектирования ориентированы на создание и преобразование разных объектов и предметов. *Объект проектирования* — это среда или процесс, в контексте которых находится предмет. *Предмет проектирования* — это предполагаемый продукт, образ которого первоначально представлен в проекте. Это то, созданию чего посвящена проектная деятельность. Объект и предмет проектирования соотносятся между собой как общее и частное.

2. Процесс проектирования.

Проектирование—это комплекс работ с целью получения описаний нового или модернизируемого технического объекта, достаточных для реализации или изготовления объекта в заданных условиях. Объектами проектирования могут быть изделия (например, обрабатывающий центр, двигатель внутреннего сгорания, ЭВМ) или процессы (например, технологические, вычислительные). Комплекс проектных работ включает в себя теоретические и экспериментальные исследования, расчеты, конструирование.

Проектирование, осуществляемое человеком при взаимодействии с ЭВМ, называют *автоматизированным.* Степень автоматизации может быть различной и оценивается долей s проектных работ, выполняемых на ЭВМ без участия человека. При s=0 проектирование называют *неавтоматизированным,* при s=1—*автоматическим.*

Автоматизированное проектирование осуществляется в рамках САПР. В соответствии с ГОСТом *система автоматизированного проектирования*—это организационно-техническая система, состоящая из комплекса средств автоматизации проектирования (АП), взаимодействующего с подразделениями проектной организации, и выполняющая автоматизированное проектирование.

Проектирование делится на стадии, этапы и процедуры. При проектировании сложных объектов выделяют стадии

- научно-исследовательских работ (НИР)
- опытно-конструкторских работ (ОКР)
- технического проекта
- рабочего проекта
- испытаний опытного образца.

Стадию *НИР* во многих случаях можно разделить на стадии

- предпроектных исследований
- технического задания
- технического предложения.

На этих стадиях последовательно изучаются потребности в получении новых изделий с заданным целевым назначением, исследуются физические, информационные, конструктивные и технологические принципы построения изделий. Далее исследуются возможности реализации этих принципов, прогнозируются возможные значения характеристик и параметров объектов. *Результатом НИР является формулировка технического задания (ТЗ) на разработку нового объекта.*

На стадии *ОКР* разрабатывается эскизный проект изделия, проверяются, конкретизируются и корректируются принципы и положения, установленные на стадии НИР.

На стадии *технического проекта* принимаются подробные технические решения и прорабатываются все части проекта.

На стадии *рабочего проекта* создается полный комплект конструкторско-технологической документации, достаточный для изготовления объекта.

На стадии *испытаний опытного образца* (или пробной партии при крупносерийном производстве) получают результаты, позволяющие выявить возможные ошибки и недоработки проекта, принимаются меры к их устранению, после чего документация передается на предприятия, выделенные для серийного производства изделий.

Проектирование разделяется также на этапы. Используются при этом следующие понятия. *Проектное решение*—описание объекта или его составной части, достаточное для рассмотрения и принятия заключения об окончании проектирования или путях его продолжения. *Проектная процедура*—часть проектирования, заканчивающаяся получением проектного решения. Примерами проектных процедур служат синтез функциональной схемы устройства, оптимизация параметров функционального узла, трассировка межсоединений на печатной плате и т. п. *Этап проектирования* —это условно выделенная часть проектирования, сводящаяся к выполнению одной или нескольких проектных процедур, объединенных по признаку принадлежности получаемых проектных решений к одному иерархическому уровню и (или) аспекту описаний.

На любой стадии или этапе проектирования можно выявить ошибочность или неоптимальность ранее принятых решений и, следовательно, необходимость или целесообразность их пересмотра. Подобные возвраты характерны для проектирования и обусловливают его *итерационный характер*. Может быть также выявлена необходимость корректировки ТЗ. Вводят понятия *процедур внешнего и внутреннего проектирования.* К внешнему проектированию относят процедуры формирования или корректировки технического задания, а к внутреннему—процедуры реализации сформированного ТЗ. Тогда можно сказать, что происходит чередование процедур внешнего и внутреннего проектирования, что особенно характерно для ранних стадий (НИР, ОКР). При этом различают нисходящее (сверху вниз) и восходящее (снизу вверх) проектирование. При *нисходящем проектировании* задачи высоких иерархических уровней решаются прежде, чем задачи более низких иерархических уровней. При *восходящем проектировании* последовательность противоположная. Функциональное проектирование сложных систем чаще всего является нисходящим, конструкторское проектирование—восходящим.

3. Формальная логика.

Формальная логика — конструирование и исследование правил преобразования высказываний, сохраняющих их истинностное значение безотносительно к содержанию входящих в эти высказывания понятий. Формальная логика, в отличие

от неформальной, организована как формальная система, обладающая высоким уровнем абстракции и чётко определёнными методами, правилами и законами[1]. Формальная логика как наука занимается выводом нового знания на основе ранее известного без обращения в каждом конкретном случае к опыту, а применением законов и правил мышления. Начальной ступенью формальной логики можно считать традиционную логику, а математическую логику — её следующей ступенью, использующей математические методы, символический аппарат и логические исчисления

4. Процесс образования понятия.

Простейшей категорией формальной логики является **понятие - оно *фиксирует мысль о предмете.*** Обычно понятие определяется через более широкое понятие путем добавления к родовому признаку видового различия.

Для человека, занимающегося научными изысканиями, постоянно необходимо получать новую информацию. Для этого ученый читает множество литературы по избранному предмету, ведет наблюдение, делает опыты. Однако вся эта деятельность была бы бесполезной, если бы не приводила к образованию новых понятий. Иными словами, полученная информация в таком случае так и осталась бы лишь информацией, не облеченной в форму, пригодную для закрепления и передачи.

Именно поэтому необходимо знать о приемах образования понятий. Такими приемами являются: абстрагирование, анализ, синтез, сравнение и обобщение.

Абстрагирование — это прием образования понятий, при котором необходимо отвлечься от ряда несущественных признаков предмета, отринуть их и оставить лишь существенные.

В процессе абстрагирования значительную роль играет сравнение.

Анализ — это мысленное дробление предмета, процесса или явления на составные части с целью установления взаимодействия этих частей и взаимосвязей между ними, а также выявления происходящих внутри исследуемого объекта процессов.

Анализ необходим для получения отражения уже существующего понятия.

Синтез — это мысленная сборка составных частей предмета, явления или процесса воедино.

Синтез — это процесс, обратный анализу, и обычно используется, когда последний уже проведен. Зачастую мысленному синтезу предшествует, если речь идет о предмете, практическая сборка данного предмета со строгим соблюдением последовательности постановки составных частей.

Синтез применяется для создания новых понятий на основе уже существующих, подвергнутых синтезу, или выявления неточностей в понятии, а также внесения в эти понятия изменений.

Сравнение — это мысленное установление сходства или различия предметов по существенным или несущественным признакам.

Обобщение — мысленное объединение группы предметов в новый ряд или добавление одного предмета в уже существующий на основе присущих этим предметам признаков.

Сравнение и обобщение позволяют достичь большей точности в суждениях, отделить одно от другого или, наоборот, объединить несколько предметов в одну группу (класс). Как факультативный признак, способствуют лучшему усвоению информации человеческим мозгом.

Все логические приемы образования понятий имеют важнейшее значение. Они связаны между собой, их невозможно представить один без другого. Часто применяются вместе или предшествуют один другому.

5. Диалектическая логика.

Диалектическая логика — философский раздел марксизма. В широком смысле понималась как систематически развёрнутое изложение диалектики мышления:диалектическое изложение науки о научно-теоретическом

мышлении («диалектики как логики»), которая тем самым является и научной теорией познания объективного мира. В узком смысле понималась как логическая дисциплина о формах правильных рассуждений.

Предмет диалектической логики — мышление. Диалектическая логика имела своей целью развернуть его изображение в необходимых его моментах и притом в независящей ни от воли, ни от сознания последовательности, а также утвердить свой статус как логической дисциплины.

6. Содержательно-генетическая логика.

СОДЕРЖАТЕЛЬНО-ГЕНЕТИЧЕСКАЯ ЛОГИКА - логико-эпистемологическая концепция, которая разрабатывалась в 1950-1960 в Московском логическом (в дальнейшем - методологическом) кружке.

Такая логика полагалась содержательной в противовес формальной. С другой стороны, логическая разработка диалектики в советской философии оценивалась еще более негативно и критически. Вопреки всей европейской традиции, логика рассматривалась как эмпирическая наука. Предметом эмпирических исследований объявлялись способы мышления, зафиксированные в научных текстах. Таким образом, основной эмпирией исследования становился текст. Тезис об эмпиричности был вместе с тем способом преодоления схоластичности существовавшей диалектики и нормативной бессодержательности формальной логики.

В целом концепция С.-Г.Л., выдвинутая Щедровицким, представляла собой разработку трех основных положений:

1) В основе концепции лежала гипотеза о "двух-плоскостной" структуре мышления, включающей плоскость "объективного содержания" и плоскость "знаковой формы". Формальная логика, начиная с Аристотеля, строилась на основе принципа параллелизма формы и содержания мышления, т.е. на основе предположения, что а) каждому элементу знаковой формы языковых выражений соответствует определенный субстанциальный элемент содержания и б) способ связи элементов содержания в точности соответствует способу связи элементов знаковой формы. На основе этого принципа в формальной логике производилась редукция содержания к плоскости знаковой формы. Такой прием ограничивал реальное содержание аристотелевской логики атрибутивными свойствами объектов. Другие типы содержания, не сводящиеся к атрибутивным свойствам, не могли быть в ней адекватно представлены. В противовес этому, С.-Г.Л. выдвигала принцип непараллелизма формы и содержания. Реализация этого принципа требовала описывать и изображать плоскость содержания (оперирование с объектами) отдельно от плоскости знаковой формы, а также описывать механизм связи между содержанием и формой. Таким образом, предметом критики выступала не "формальность" классической логики сама по себе (т.е. ее структурность и нормативность), а редукция содержания мышления к якобы "всеобщим" и "универсальным" логическим формам.

2) В качестве основного объяснительного принципа по отношению к мышлению выдвигалась категория "деятельности". Само мышление рассматривалось двояко: как фиксированное знание и как деятельность по его получению. Принцип деятельности требовал рассматривать мышление не по содержанию предметного сознания, а структурно-процессуально, т.е. исследовать и описывать, с одной стороны, процедуры и операции мышления, а с другой - типологически обобщенную структуру знаний. Таким образом, концепция С.-Г.Л. смыкалась с программой исследования мышления как деятельности. Деятельностный подход позволял сохранить установку на содержательность, избегая психологизма: содержание сознания выносилось "за скобки", а деятельностное содержание трактовалось операционально. "Воспроизвести содержание мышления" означало воспроизвести схему оперирования с объектами, схему их сопоставлений и т.п. Соответственно этому отвергалась и трактовка мышления как "отражения". Основной мыслительной связью объявлялось не отражение, а "замещение" исходного практического оперирования с объектами знаками и оперированием с ними.

Мышление представлялось как деятельность в иерархической структуре последовательно надстраивающихся друг над другом плоскостей знакового замещения, каждая из которых задавала определенные системы оперирования.

3) Общая теоретическая задача С.-Г.Л. состояла в том, чтобы на основе анализа единичных эмпирически заданных текстов проанализировать и воспроизвести в форме "исторической теории" мышление вообще, мышление как таковое, как один органический предмет. При этом мышление изначально рассматривалось в синтетическом единстве с языком, как языковое мышление. Раздельный анализ мышления и языка признавался неэффективным. Генетичность новой логики определялась, с одной стороны, ориентацией на исследование процессов развития мышления, а с другой стороны, использованием метода восхождения от абстрактного к конкретному.

7. Логическая схема проектирования.

Проектирование – процесс, позволяющий провести некую техническую идею до её инженерной модели. Результатом этого процесса является проект, который представляет из себя, как правило, графическую часть (чертежи, схемы) и пояснительную записку (описание назначения изделия, функции, технические характеристики и т.д.).

Алгоритм проектирования – совокупность предписаний, необходимых для выполнения проектирования. Алгоритм проектирования может быть общим (для определенного класса объектов) и специальным (для одного объекта). Под выполнением проектирования понимается нахождение результата проектирования.

Результат проектирования – проектное решение (совокупность проектных решений), удовлетворяющее заданным требованиям, необходимое для создания объекта проектирования. В заданные требования должны быть включены требования к форме представления проектного решения.

Проектное решение – промежуточное или конечное описание объекта проектирования, необходимое и достаточное для рассмотрения и определения дальнейшего направления или окончания проектирования.

Типовое проектное решение – уже существующее проектное решение, используемое при проектировании.

Цель процесса проектирования состоит, прежде всего, в том, чтобы на основе исходной информации, получаемой в процессе проектирования, разработать техническую документацию для изготовления объекта проектирования. Проектирование включает в себя разработку технического задания (ТЗ), отражающего потребности, и реализацию ТЗ в виде проектной документации.

Проектирование, по существу, представляет собой процесс управления с обратной связью. Техническое задание формирует входы, которые сравниваются с результатами проектирования, и если они не совпадают, цикл проектирования повторяется вновь до тех пор, пока отклонение от заданных технических требований не окажется в допустимых пределах.

Проектная процедура соответствует части проектной подсистемы, в результате выполнения которой принимается некоторое проектное решение. Она состоит из элементарных проектных операции, имеет твердо установленный порядок их выполнения и направлена на достижение локальной цели в процессе проектирования.

Под проектной операцией понимают условно выделенную часть проектной процедуры или элементарное действие, совершаемое конструктором в процессе проектирования. Примерами проектных процедур могут служить процедуры разработки кинематической или компоновочной схемы станка, технологии обработки изделий и т.п., а примерами проектных операций – расчет припусков, решение какого-либо уравнения и т.п.

8. Логические отношения.

Так как все предметы находятся во взаимодействии и взаимообусловленности, то и понятия, отражающие данные предметы, также находятся в определенных отношениях.

Конкретные виды отношений устанавливаются в зависимости от содержания и объема понятий, которые сравниваются.

Если понятия не имеют общих признаков, далеки друг от друга по свое-му содержанию, то они называются несравнимыми. Например, «симфониче-ская музыка» и «кассационная жалоба», «процессуальные акты предварительного расследования» и «общая тетрадь».

Сравнимыми называются понятия, отражающие некоторые общие суще-ственные признаки предмета или класса однородных предметов. Например, «юрист» и «адвокат», «взятка» и «кража».

В логических отношениях могут находиться только сравнимые понятия. В зависимости от того, как соотносятся их объемы, понятия делятся на две группы: совместимые и несовместимые.

Совместимые - это такие понятия, объемы которых совпадают полно-стью или частично. Несовместимые - это понятия, объемы которых не сов-падают ни в одном элементе, но которые могут быть включены частично или полностью в объем общего для них понятия. На представленной схеме по-казаны виды совместимых и несовместимых понятий.

Отношения между понятиями принято иллюстрировать при помощи кругов Эйлера (круговых схем).

В отношениях равнозначности находятся совместимые понятия, объемы которых полностью совпадают. В таких понятиях мыслится один и тот же предмет или класс однородных предметов. Однако содержание этих понятий различно, так как каждое из них отражает только определенную сторону (существенный признак) данного предмета или класса однородных предметов.

В отношении пересечения находятся совместимые понятия, у которых объемы частично совпадают. Частично совпадает и содержание данных понятий.

В отношении подчинения находятся совместимые понятия, объем одно-го из которых полностью входит в объем другого, составляя его часть.

Объем первого понятия шире объема второго понятия: кроме кражи личного имущества граждан в него входит также кража государственного, кооперативного имущества.

Из двух понятий, находящихся в отношении подчинения, понятие с большим объемом (подчиняющее) является родовым, или родом по отношению к понятию с меньшим объемом (подчиненному), а последнее по отношению к первому называется видовым, или видом. Родовидовые отноше-ния лежат в основе логических операций ограничения и обобщения понятий, деления объема понятий и некоторых видов определения.

Перейдем к рассмотрению несовместимых понятий.

При иллюстрации отношений между несовместимыми понятиями возни-кает потребность во введении более широкого по объему понятия, которое включало бы объемы несовместимых понятий.

В отношении соподчинения находятся два или более непересекающихся понятий, принадлежащих общему родо-вому понятию.

Соподчиненные понятия В и С - это виды одного рода А, у них общий родовой признак, но видовые признаки различны.

В отношении противоположности находятся понятия, которые являются видами одного и того же рода, и при этом одно из них содержит какие-то признаки, а другое эти признаки отрицает и заменяет противоположными при-знаками.

В отношении противоречия находятся такие два поня-тия, которые являются видами одного и того же рода, и при этом одно понятие указывает на некоторые признаки, а другое эти признаки отрицает, исключает, не заменяя их никакими другими признаками.

9. Определение задач проектирования.

На стадии проектирования проектировщик должен проделать следующую работу:

1. Обследовать предметную область автоматизации.

2. Определить объекты и перечень их атрибутов, для каждого объекта выделить первичные ключи и провести нормализацию.

3. Установить все связи между объектами. Начертить схему проекта со всеми объектами и связями.

4. Выработать технологию обслуживания, т.е. определить порядок сбора, хранения данных в БД частоту и форматы ввода-вывода данных, правила работы всех групп пользователей.

5. Выбрать компьютер и инструментальные средства (конкретную СУБД) для реализации.

6. Проверить корректность проекта. Проект должен адекватно, на требуемом уровне детальности, отображать предметную область.

10. Определение логической схемы проектирования (методологии проектирования).

Методология логического проектирования

Подразделяется на следующие этапы:

1. Построение и проверка локальной логической модели для отдельных представлений каждого из пользователей.

2. Создание и проверка глобальной логической модели данных.

На первом этапе, на основе концептуальной модели данных строится локальная логическая модель. Данный этап включает следующие моменты:

1. Локальная концептуальная модель преобразуется в локальную логическую.

2. Исходя из структуры локальной логической модели определяются наборы отношений.

3. Проверка модели с помощью правил нормализации.

4. Проверка модели в отношении транзакции пользователя.

5. Создание уточненной диаграммы “сущность – связь”.

6. Определение требований поддержки целостности данных.

7. Обсуждение разработанной локальной логической модели с конечным пользователем.

11. Решение задач проектирования.

Одним из путей предсказания поведения проектируемых систем является путь создания математических моделей и последующего проведения исследования систем на этих моделях. Построение или проектирование систем, удовлетворяющих заранее заданным свойствам, можно осуществить, когда имеются управляющие переменные, при помощи которых можно влиять на поведение проектируемой системы.

1. Понятие системного подхода и системного анализа.

Основной общий принцип системного подхода заключается в рассмотрении частей явления или сложной системы с учетом их взаимодействия. *Системный подход* включает в себя выявление структуры системы, типизацию связей, определение атрибутов, анализ влияния внешней среды. Системный подход рассматривают как направление научного познания и социальной политики. Он является базой для обобщающей дисциплины «Теория систем» (другое используемое название - «Системный анализ»). *Теория систем* - дисциплина, в которой конкретизируются положения системного подхода; она посвящена исследованию и проектированию сложных экономических, социальных, технических систем, чаще всего слабоструктурированных. Характерными примерами таких систем являются производственные системы. При проектировании систем цели достигаются в

многошаговых процессах принятия решений. Методы принятия решений часто выделяют в самостоятельную дисциплину, называемую «Теория принятия решений».

В технике дисциплину, в которой исследуются сложные технические системы, их проектирование и которая аналогична теории систем, чаще называют *системотехникой.* Предметом системотехники являются, во-первых, организация процесса создания, использования и развития технических систем, во-вторых, методы и принципы их проектирования и исследования. В системотехнике важно уметь сформулировать цели системы и организовать ее рассмотрение с позиций поставленных целей. Тогда можно отбросить малозначимые части при проектировании и моделировании, перейти к постановке оптимизационных задач.

Системы автоматизированного проектирования и управления относятся к числу наиболее сложных современных искусственных систем. Их проектирование и сопровождение невозможны без системного подхода. Поэтому идеи и положения системотехники входят составной частью в дисциплины, посвященные изучению современных автоматизированных систем и технологий их применения.

Интерпретация и конкретизация системного подхода имеют место в ряде известных подходов с другими названиями, которые также можно рассматривать как компоненты системотехники. Таковы структурный, блочно-иерархический, объектно-ориентированный подходы.

При *структурном подходе,* как разновидности системного, требуется синтезировать варианты системы из компонентов (блоков) и оценивать варианты при их частичном переборе с предварительным прогнозированием характеристик компонентов.

Блочно-иерархический подход к проектированию использует идеи декомпозиции сложных описаний объектов и соответственно средств их создания на иерархические уровни и аспекты, вводит понятие стиля проектирования (восходящее и нисходящее), устанавливает связь между параметрами соседних иерархических уровней.

Ряд важных структурных принципов, используемых при разработке информационных систем и прежде всего их программного обеспечения (ПО), выражен в *объектно-ориентированном подходе* к проектированию. Такой подход имеет следующие преимущества в решении проблем управления сложностью и интеграции ПО: 1) вносит в модели приложений большую структурную определенность, распределяя представленные в приложении данные и процедуры между классами объектов; 2) сокращает объем спецификаций благодаря внедрению в описания иерархии объектов и отношений наследования между свойствами объектов разных уровней иерархии; 3) уменьшает вероятность искажения данных вследствие ошибочных действий за счет ограничения доступа к определенным категориям данных в объектах. Описание в каждом классе объектов допустимых обращений к ним и принятый форматов сообщений облегчает согласование и интеграцию ПО.

Для всех подходов к проектированию сложных систем характерны также следующие особенности.

- Структуризация процесса проектирования, выражаемая декомпозицией проектных задач и документации, выделением стадий, этапов, проектных процедур. Эта структуризация является сущностью блочно-иерархического подхода к проектированию.
- Итерационный характер проектирования.
- Типизация и унификация проектных решений и средств проектирования.

Существуют различные точки зрения на содержание понятия «системный анализ» и область его применения. Изучение различных определений системного анализа позволяет выделить четыре его трактовки.

Первая трактовка рассматривает системный анализ как один из конкретных методов выбора лучшего решения возникшей проблемы, отождествляя его, например, с анализом по критерию стоимость — эффективность.

Такая трактовка системного анализа характеризует попытки обобщить наиболее разумные приемы любого анализа (например, военного или экономического), определить общие закономерности его проведения.

В первой трактовке системный анализ — это, скорее, «анализ систем», так как акцент делается на объекте изучения (системе), а не на системности рассмотрения (учете всех важнейших факторов и взаимосвязей, влияющих на решение проблемы, использование определенной логики поиска лучшего решения и т.д.)

В ряде работ, освещающих те или иные проблемы системного анализа, слово «анализ» употребляется с такими прилагательными, как количественный, экономический, ресурсный, а термин «системный анализ» применяется значительно реже.

Согласно второй трактовке системный анализ — это конкретный метод познания (противоположность синтезу).

Третья трактовка рассматривает системный анализ как любой анализ любых систем (иногда добавляется, что анализ на основе системной методологии) без каких-либо дополнительных ограничений на область его применения и используемые методы.

Согласно четвертой трактовке системный анализ — это вполне конкретное теоретико-прикладное направление исследований, основанное на системной методологии и характеризующееся определенными принципами, методами и областью применения. Он включает в свой состав как методы анализа, так и методы синтеза, кратко охарактеризованные нами ранее.

Нам представляется правильной четвертая трактовка8, наиболее адекватно отражающая направленность системного анализа и совокупность используемых им методов.

Итак, системный анализ — это совокупность определенных научных методов и практических приемов решения разнообразных проблем, возникающих во всех сферах целенаправленной деятельности общества, на основе системного подхода и представления объекта исследования в виде системы. Характерным для системного анализа является то, что поиск лучшего решения проблемы начинается с определения и упорядочения целей деятельности системы, при функционировании которой возникла данная проблема. При этом устанавливается соответствие между этими целями, возможными путями решения возникшей проблемы и потребными для этого ресурсами.

Системный анализ характеризуется главным образом упорядоченным, логически обоснованным подходом к исследованию проблем и использованию существующих методов их решения, которые могут быть разработаны в рамках других наук.

Целью системного анализа является полная и всесторонняя проверка различных вариантов действий с точки зрения количественного и качественного сопоставления затраченных ресурсов с получаемым эффектом.

Системный анализ, по существу, является средством установления рамок для систематизированного и более эффективного использования знаний, суждений и интуиции специалистов; он обязывает к определенной дисциплине мышления.

2. Документ. Электронный документ. Информационная система.

Документ является основным способом представления информации. ***Электронный документ*** — это бумажный документ, введённый в*компьютер* для обработки. Финансовые электронные документы могут снабжаться *электронной подписью*. Электронные документы бывают структурированными, и тогда они находятся в базах данных, и неструктурированными, содержащими тексты на естественном языке.

В общем же принято считать под *электронным документом* (ЭД) структурированный информационный *объект*, в соответствие которому может быть поставлена совокупность файлов, хранящихся на накопителе компьютера. Необходимым признаком ЭД является "регистрационная карточка", состоящая из реквизитов документа, содержащих перечень необходимых данных о нём. Согласно законодательству,

электронным является документ, в котором *информация* представлена в электронно-цифровой форме.

Электронные документы можно разделить на два основных типа: неформализованные (произвольные) и служебные (официальные). Неформализованный *электронный документ* — это любое сообщение, записка, текст, записанный на машинном носителе. Под служебным *электронным документом* понимается записанное на машинном носителе электронное сообщение, реквизиты которого оформлены в соответствии с нормативными требованиями.

Электронные документы *по* сравнению с бумажными обладают следующими преимуществами:

- более низкая стоимость и время передачи электронного документа из одного места в другое;
- более низкая стоимость и время тиражирования ЭД;
- более низкая стоимость архивного хранения ЭД;
- возможность контекстного поиска;
- новые возможности защиты документов;
- упрощение подготовки ЭД в сочетании с широкими возможностями;
- принципиально новые возможности представления ЭД. Документ может иметь динамическое содержание (например, аудио-, видеовставки).

Информационная система (сокр. ИС) — система, предназначенная для хранения, поиска и обработки информации и соответствующие организационные ресурсы (человеческие, технические, финансовые и т. д.), которые обеспечивают и распространяют информацию.

Информационная система предназначена для своевременного обеспечения надлежащих людей надлежащей информацией, то есть для удовлетворения конкретных информационных потребностей в рамках определенной предметной области, при этом результатом функционирования информационных систем являетсяинформационная продукция — документы, информационные массивы, базы данных и информационные услуги.

Понятие информационной системы интерпретируют по-разному, в зависимости от контекста.

Достаточно широкое понимание информационной системы подразумевает, что её неотъемлемыми компонентами являются данные, техническое и программное обеспечение, а также персонал и организационные мероприятия. Широкотрактует понятие «информационной системы» федеральный закон Российской Федерации«Об информации, информационных технологиях и о защите информации», подразумевая под информационной системой совокупность содержащейся в базах данныхинформации и обеспечивающих её обработку информационных технологий и технических средств[7].

Среди российских ученых в области информатики, наиболее широкое определение ИС дает М. Р. Когаловский, по мнению которого в понятие информационной системы помимо данных, программ, аппаратного обеспечения и людских ресурсов следует также включать коммуникационное оборудование, лингвистические средства и информационные ресурсы, которые в совокупности образуют систему, обеспечивающую «поддержку динамической информационной модели некоторой части реального мира для удовлетворения информационных потребностей пользователей».

Более узкое понимание информационной системы ограничивает её состав данными, программами и аппаратным обеспечением. Интеграция этих компонентов позволяет автоматизировать процессы управления информацией и целенаправленной деятельности конечных пользователей, направленной на получение, модификацию и хранение информации. Так, российский стандарт ГОСТ РВ 51987 подразумевает под ИС «автоматизированную систему, результатом функционирования которой является

представление выходной информации для последующего использования». ГОСТ Р 53622-2009 использует термин*информационно-вычислительная система* для обозначения совокупности данных (или баз данных), систем управления базами данных и прикладных программ, функционирующих на вычислительных средствах как единое целое для решения определенных задач.

В деятельности организации информационная система рассматривается как программное обеспечение, реализующее деловую стратегию организации. При этом хорошей практикой является создание и развертывание единой корпоративной информационной системы, удовлетворяющей информационные потребности всех сотрудников, служб и подразделений организации. Однако на практике создание такой всеобъемлющей информационной системы слишком затруднено или даже невозможно, вследствие чего на предприятии обычно функционируют несколько различных систем, решающих отдельные группы задач: управление производством,финансово-хозяйственная деятельность, электронный документооборот и т. д. Часть задач бывает «покрыта» одновременно несколькими информационными системами, часть задач — вовсе не автоматизирована. Такая ситуация получила название «лоскутной автоматизации» и является довольно типичной для многих предприятий.

3. Информационная технология.

Основной составляющей частью автоматизированной информационной системы является информационная технология (ИТ), развитие которой тесно связано с развитием и функционированием ИС.

Понятие "***технология***"в переводе с греческого означает искусство, мастерство, умение. Технология, как процесс, означает последовательность ряда действий с целью переработки чего-либо. Технологический процесс реализуется различными средствами и методами.

Процесс материального производства предполагает обработку ресурсов с целью получения материальных продуктов (товаров). Если речь идет об информационных технологиях, то роль ресурсов играют данные.

Информационная технология – процесс, использующий совокупность средств методов сбора, обработки и передачи первичной информации для получения информации нового качества о состоянии объекта, т.е. информационного продукта.

Информационный продукт используется, в частности, для принятия решений. Существует *разница между понятиями "информационная система" и "информационная технология"*.

Информационная технология является процессом, состоящим из четко регламентированных операций по преобразованию информации (сбор данных, их регистрация, передача, хранение, обработка, использование).

Компьютерная информационная система является человеко-машинной системой обработки информации с целью организации, хранения и передачи информации. Например, технология, работающая с текстовым редактором, не является информационной системой.

Информационные технологии состоят из этапов, каждый из них включает операции, а последние состоят из элементарных действий, таких как нажатие какой-нибудь клавиши, выбор позиции в меню и т.д.

В информационных технологиях экономических систем широкое распространение получили офисные программы, включающие: табличные процессоры; текстовые процессоры; СУБД; интегрированные пакеты и пр.

Информационные технологии прошли несколько этапов. Каждый этап определяется техникой, программными продуктами, которые используются, т.е. уровнем научно-технического прогресса в этой области.

В настоящее время используется понятие "***новая информационная технология***". Это понятие предполагает:

1. Использование персональных компьютеров и сетей ПК.
2. Наличие коммуникационных средств.
3. Наличие диалоговой (интерактивной) работы с компьютером.
4. Наличие интеграционного подхода.
5. Гибкость процессов изменения данных и постановок задач.
6. Органическое "встраивание" компьютеров в существующую на предприятиях технологию управления.

Основная цель автоматизированной информационной технологии – получать посредством переработки первичных данных информацию нового качества, на основе которой вырабатываются оптимальные управленческие решения. Это достигается за счет интеграции информации, обеспечения ее актуальности и непротиворечивости, использования современных технических средств для внедрения и функционирования качественно новых форм информационной поддержки деятельности аппарата управления.

Информационная технология справляется с существенным увеличением объемов перерабатываемой информации и ведет к сокращению сроков ее обработки. ИТ является наиболее важной составляющей процесса использования информационных ресурсов в управлении. Автоматизированные информационные системы для информационной технологии – это основная среда, составляющими элементами которой являются средства и способы для преобразования данных. ***Информационная технология*** представляет собой процесс, состоящий из четко регламентированных правил выполнения операций над информацией, циркулирующей в ИС.

1.2 Лекция №3, 4 (4 часа)

Тема: «Виды моделей компонентов информационных систем»

1.2.1 Вопросы лекции:

1. Модели жизненного цикла информационных систем.
2. Каскадная модель.
3. Инкрементная модель.
4. Эволюционная модель.

1.2.2 Краткое содержание вопросов:

1. Модели жизненного цикла информационных систем.

Методология проектирования информационных систем описывает процесс создания и сопровождения систем в виде *жизненного цикла* (ЖЦ) ИС, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых *работ*, получаемые результаты, методы и средства, необходимые для выполнения *работ*, роли и ответственность участников и т.д. Такое формальное описание ЖЦ ИС позволяет спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом.

Жизненный цикл ИС можно представить как ряд событий, происходящих с системой в процессе ее создания и использования.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления. *Модель жизненного цикла* - структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

В настоящее время известны и используются следующие *модели жизненного цикла*:

- ***Каскадная модель*** (рис. 1) предусматривает последовательное выполнение всех *этапов проекта* в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе.
- ***Поэтапная модель с промежуточным контролем*** (рис. 2). Разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.
- ***Спиральная модель*** (рис.3). На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.Особое внимание уделяется начальным этапам разработки - анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов (*макетирования*).

Рисунок - 1**.** Каскадная модель ЖЦ ИС

Рисунок - 2**.** Поэтапная модель с промежуточным контролем

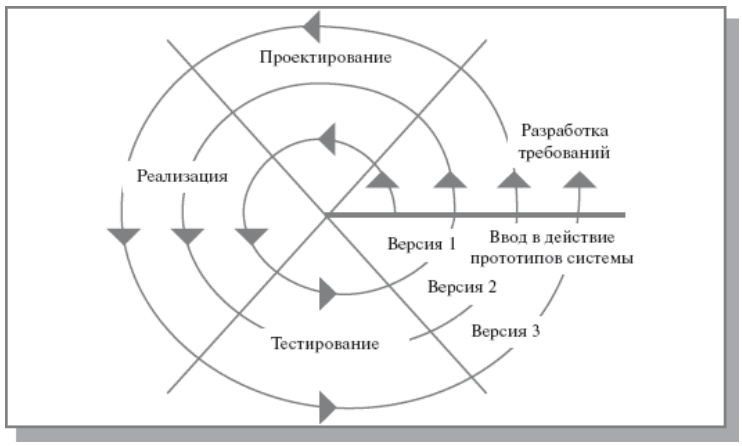

Рисунок - 3. Спиральная модель ЖЦ ИС

На практике наибольшее распространение получили две основные *модели жизненного цикла*:

- *каскадная модель* (характерна для периода 1970-1985 гг.);
- *спиральная модель* (характерна для периода после 1986.г.).

В ранних проектах достаточно простых ИС каждое *приложение* представляло собой единый, функционально и информационно независимый блок. Для разработки такого типа приложений эффективным оказался каскадный способ. Каждый этап завершался после полного выполнения и документального оформления всех предусмотренных *работ*.

Можно выделить следующие положительные стороны применения каскадного подхода:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении относительно простых ИС, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным недостатком этого подхода является то, что реальный процесс создания системы никогда полностью не укладывается в такую жесткую схему, постоянно возникает потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ИС оказывается соответствующим *поэтапной модели с промежуточным контролем*.

Однако и эта схема не позволяет оперативно учитывать возникающие изменения и уточнения требований к системе. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа *работ*, а общие требования к ИС зафиксированы в виде технического задания на все время ее создания. Таким образом, пользователи зачастую получают систему, не удовлетворяющую их реальным потребностям.

Спиральная модель ЖЦ была предложена для преодоления перечисленных проблем. На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным *требованиям заказчика* и доводится до реализации.

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального *цикла* - *определение* момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов *жизненного цикла*, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Несмотря на настойчивые рекомендации компаний - вендоров и экспертов в области проектирования и разработки ИС, многие компании продолжают использовать *каскадную модель* вместо какого-либо варианта итерационной модели. Основные причины, *по* которым *каскадная модель* сохраняет свою популярность, следующие:

1. **Привычка** - многие ИТ-специалисты получали образование в то время, когда изучалась только *каскадная модель*, поэтому она используется ими и в наши дни.
2. **Иллюзия снижения рисков** участников проекта (заказчика и исполнителя). *Каскадная модель* предполагает разработку законченных продуктов на каждом этапе: технического задания, *технического проекта*, программного продукта и пользовательской документации. Разработанная документация позволяет не только определить требования к продукту следующего этапа, но и определить обязанности сторон, объем работ и сроки, при этом окончательная оценка сроков и стоимости проекта производится на начальных этапах, после завершения обследования. Очевидно, что если требования к информационной системе меняются в ходе реализации проекта, а качество документов оказывается невысоким (требования неполны и/или противоречивы), то в действительности использование *каскадной модели* создает лишь иллюзию определенности и на деле увеличивает риски, уменьшая лишь ответственность участников проекта. При формальном подходе *менеджер проекта* реализует только те требования, которые содержатся в спецификации, опирается на документ, а не на реальные потребности бизнеса. Есть два основных типа контрактов на разработку ПО. Первый тип предполагает выполнение определенного объема работ за определенную сумму в определенные сроки (fixedprice). Второй тип предполагает повременную оплату работы (timework). Выбор того или иного типа контракта зависит от степени определенности задачи. *Каскадная модель* с определенными этапами и их результатами лучше приспособлена для заключения контракта с оплатой по результатам работы, а именно этот тип контрактов позволяет получить полную *оценку стоимости* проекта до его завершения. Более вероятно заключение контракта с повременной оплатой на небольшую систему, с относительно небольшим весом в структуре затрат предприятия. Разработка и внедрение интегрированной информационной системы требует существенных финансовых затрат, поэтому используются контракты с фиксированной ценой, и, следовательно, *каскадная модель* разработки и внедрения. *Спиральная модель* чаще применяется при разработке информационной системы силами собственного отдела ИТ предприятия.
3. **Проблемы внедрения** при использовании итерационной модели. В некоторых областях *спиральная модель* не может применяться, поскольку невозможно использование/тестирование продукта, обладающего неполной функциональностью (например, военные разработки, атомная энергетика и т.д.). Поэтапное итерационное внедрение информационной системы для бизнеса возможно, но сопряжено с организационными сложностями (перенос данных, интеграция систем, изменение бизнес-процессов, *учетной политики*, обучение пользователей). Трудозатраты при поэтапном итерационном внедрении оказываются значительно выше, а управление проектом требует настоящего искусства. Предвидя указанные сложности, заказчики выбирают *каскадную модель*, чтобы "внедрять систему один раз".

Каждая из стадий создания системы предусматривает выполнение определенного объема *работ*, которые представляются в виде *процессов ЖЦ*. *Процесс* определяется как совокупность взаимосвязанных действий, преобразующих входные данные в выходные. Описание каждого процесса включает в себя перечень решаемых задач, исходных данных и результатов.

2. Каскадная модель.

Каскадная модель. Каскадная модель жизненного цикла («модель водопада», англ. waterfallmodel) была предложена в 1970 г. Уинстоном Ройсом. Она предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Этапы проекта в соответствии с каскадной моделью:

1.Формирование требований

2.Проектирование

3.Реализация

4.Тестирование

5.Ввод в действие

6.Эксплуатация и сопровождение

Каскадные технологические подходы к разработке информационных систем задают некоторую последовательность выполнения процессов, обычно изображаемую в виде каскада. Эти подходы также иногда называют подходами на основе модели водопада.

Каскадная модель (waterfallmodel) жизненного цикла (рис.4) считается основой технологических подходов к ведению жизненного цикла. Эта модель была предложена в 1970 году У. Ройсом. Впоследствии данная модель была регламентирована множеством нормативных документов, в частности, широко известным стандартом Министерства обороны США Dod-STD-2167A и российскими стандартами серии ГОСТ 34. Принципиальными свойствами так называемой «чистой» каскадной модели являются следующее:

· фиксация требований к системе до ее сдачи заказчику;

· переход на очередную стадию проекта только после того, как будет полностью завершена работа на текущей стадии, без возвратов на пройденные стадии.

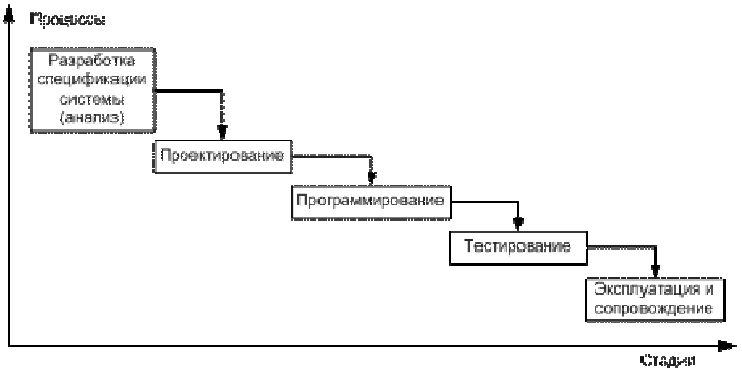

Рисунок - 4. Каскадная модель жизненного цикла.

Каждая стадия заканчивается получением некоторых результатов, которые служат в качестве исходных данных для следующей стадии. Требования к разрабатываемому программному обеспечению, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Преимущества применения каскадной модели заключаются в следующем:

· на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;

· выполняемые в логичной последовательности стадии работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадная модель может использоваться при создании информационных систем, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем, чтобы предоставить разработчикам свободу реализовать их технически как можно лучше. В эту категорию попадают, как правило, системы с высокой критичностью: сложные системы с большим количеством задач вычислительного характера, системы управления производственными процессами повышенной опасности и др.

В то же время этот подход обладает рядом недостатков, вызванных, прежде всего тем, что реальный процесс создания информационных систем никогда полностью не укладывался в такую жесткую схему. Процесс создания системы носит, как правило, итерационный характер: результаты очередной стадии часто вызывают изменения в проектных решениях, выработанных на более ранних стадиях. Таким образом, постоянно возникает потребность в возврате к предыдущим стадиям и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания информационной системы принимает вид:

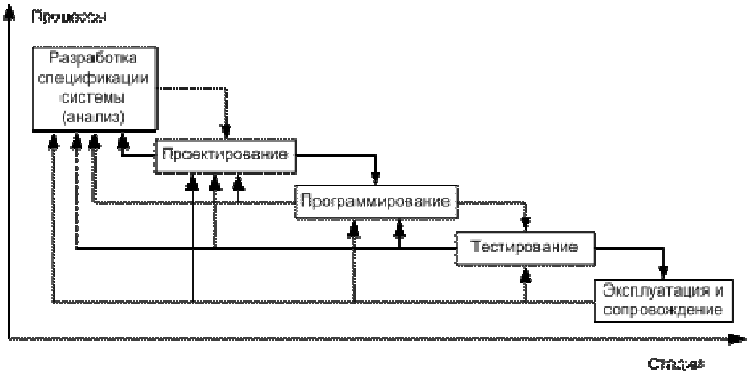

Рисунок - 5. Модель реального процесса разработки системы.

Основными недостатками каскадного подхода являются:

· позднее обнаружение проблем;

· выход из календарного графика, запаздывание с получением результатов;

· избыточное количество документации;

· невозможность разбить систему на части (весь продукт разрабатывается за один раз);

· высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей.

Практика показывает, что на начальной стадии проекта полностью и точно сформулировать все требования к будущей системе не удается. Это объясняется двумя причинами:

· пользователи не в состоянии сразу изложить все свои требования и не могут предвидеть, как они изменятся в ходе разработки;

· за время разработки могут произойти изменения во внешней среде, которые повлияют на требования к системе.

1. Инкрементная модель.

Инкрементная модель является классическим примером инкрементной стратегии конструирования. Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования (предложена Б. Боэмом как усовершенствование*каскадной модели*). Каждая линейная последовательность здесь

вырабатывает поставляемый инкремент ПО. Например, ПОдля обработки слов в 1-м инкременте (версии) реализует функции базовой обработки файлов, функции редактирования и документирования; во 2-м инкременте – более сложные возможности редактирования и документирования; в 3-м инкременте – проверку орфографии и грамматики; в 4-м инкременте – возможности компоновки страницы.

Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными). План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность.

По своей природе инкрементный процесс итеративен, но, в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт.

Схема такой модели ЖЦ ПО приведена на рис.6. Одной из современных реализаций инкрементного подхода является экстремальное программирование (ориентировано на очень малые приращения функциональности) [23].

Рисунок - 6**.** Инкрементная модель разработки программного обеспечения

Инкрементная разработка представляет собой процесс частичной реализации всей системы и медленного наращивания функциональных возможностей. Этот подход позволяет уменьшить затраты, понесенные до момента достижения уровня исходной производительности. С помощью этой модели ускоряется процесс создания функционирующей системы. Этому способствует применяемый принцип компоновки из стандартных блоков, благодаря которому обеспечивается контроль над процессом разработки изменяющихся требований.

Инкрементная модель действует по принципу каскадной модели с перекрытиями, благодаря чему функциональные возможности продукта, пригодные к эксплуатации, формируется раньше. Для этого может понадобиться полный заранее сформированный набор требований, которые выполняются в виде последовательных, небольших по размеру проектов, либо выполнение проекта может начаться с формулирования общих целей, которые затем уточняются и реализуются группами разработчиков.

Подобное усовершенствование каскадной модели одинаково эффективно при использовании как в случае чрезвычайно больших, так и небольших проектов.

Инкрементная модель описывает процесс, при выполнении которого первостепенное внимание уделяется системным требованиям, а затем их реализации в группах разработчиков. Как правило, со временем инкременты уменьшаются и реализуют каждый раз меньшее количество требований. Каждая последующая версия системы добавляет к предыдущей определенные функциональные возможности до тех пор, пока не будут реализованы все запланированные возможности. В этом случае можно уменьшить затраты, контролировать влияние изменяющихся требований и ускорить создание функциональной системы благодаря использованию метода компоновки из стандартных блоков.

Предполагается, что на ранних этапах жизненного цикла (планирование, анализ и разработка проекта) выполняется конструирование системы в целом. На этих этапах определяются относящиеся к ним инкременты и функции.

Каждый инкремент затем проходит через остальные фазы жизненного цикла: кодирование, тестирование и поставку.

Сначала выполняется конструирование, тестирование и реализация набора функций, формирующих основу продукта, или требований первостепенной важности, играющих основную роль для успешного выполнения проекта либо снижающих степень риска. Последующие итерации распространяются на ядро системы, постепенно улучшая ее функциональные возможности или рабочую характеристику. Добавление функций осуществляется с помощью выполнения существенных инкрементов с целью в комплексные удовлетворения потребностей пользователя. Каждая дополнительная функция аттестуется в соответствии с целым набором требований.

Применяя инкрементную модель при разработке проекта, для которого она подходит в достаточной мере, можно убедиться в следующих ее преимуществах:

· не требуется заранее тратить средства, необходимые для разработки всего проекта (поскольку сначала выполняется разработка и реализация основной функции или функции из группы высокого риска);

· в результате выполнения каждого инкремента получается функциональный продукт;

· заказчик располагает возможностью высказаться по поводу каждой разработанной версии системы;

· правило по принципу "разделяй и властвуй" позволяет разбить возникшую проблему на управляемые части, благодаря чему предотвращается формирование громоздких перечней требований, выдвигаемых перед командой разработчиков;

· существует возможность поддерживать постоянный прогресс в ходе выполнения проекта;

· снижаются затраты на первоначальную поставку программного продукта;

· ускоряется начальный график поставки (что позволяет соответствовать возросшим требованиям рынка);

· снижается риск неудачи и изменения требований;

· заказчики могут распознавать самые важные и полезные функциональные возможности продукта на более ранних этапах разработки;

· риск распределяется на несколько меньших по размеру инкрементов (не сосредоточен в одном большом проекте разработки);

· требования стабилизируются (посредством включения в процесс пользователей) на момент создания определенного инкремента, поскольку не являющиеся особо важными изменения отодвигаются на момент создания последующих инкрементов;

· инкременты функциональных возможностей несут больше пользы и проще при тестировании, чем продукты промежуточного уровня при поуровневой разработке по принципу "сверху-вниз"

· улучшается понимание требований для более поздних инкрементов (что обеспечивается благодаря возможности пользователя получить представление о раннее полученных инкрементов на практическом уровне);

· в конце каждой инкрементной поставки существует возможность пересмотреть риски, связанные с затратами и соблюдением установленного графика;

· использование последовательных инкрементов позволяет объединить полученный пользователями опыт в виде усовершенствованного продукта, затратив при этом намного меньше средств, чем требуется для выполнения повторной разработки;

· в процессе разработки можно ограничить количество персонала таким образом, чтобы над поставкой каждого инкремента последовательно работала одна и та же команда и все задействованные в процессе разработки команды не прекращали работу над

проектом (график распределения рабочей силы может выравниваться посредством распределения по времени объема работы над проектом);

· возможность начать построение следующей версии проекта на переходном этапе предыдущей версии сглаживает изменения, вызванные сменой персонала;

· в конце каждой инкрементной поставки существует возможность пересмотреть риски, связанные с затратами и соблюдением установленного графика;

· потребности клиента лучше поддаются управлению, поскольку время разработки каждого инкремента очень незначительно;

· поскольку переход из настоящего в будущее не происходит моментально, заказчик может привыкать к новой технологии постепенно;

· ощутимые признаки прогресса при выполнении проекта помогают поддерживать вызванное соблюдением графика "давление" на управляемом уровне.

2. Эволюционная модель.

В случае эволюционной модели система разрабатывается в виде последовательности блоков структур (конструкций). В отличие от инкрементной модели ЖЦ подразумевается, что требования устанавливаются частично и уточняются в каждом последующем промежуточном блоке структуры системы.

Использование эволюционной модели предполагает проведение исследования предметной области для изучения потребностей заказчика проекта и анализа возможности применения этой модели для реализации. Модель применяется для разработки несложных и не критических систем, для которых главным требованием является реализация функций системы. При этом требования не могут быть определены сразу и полностью. Тогда разработка системы проводится итерационно путем ее эволюционного развития с получением некоторого варианта системы - прототипа, на котором проверяется реализация требований. Иными словами, такой процесс по своей сути является итерационным, с повторяющимися этапами разработки, начиная от измененных требований и до получения готового продукта. В некотором смысле к этому типу модели можно отнести спиральную модель.

Развитием этой модели является модель эволюционного прототипирования в рамках всего ЖЦ разработки (рис. 7). В литературе она часто называется моделью быстрой разработки приложений RAD (*RapidApplicationDevelopment*). В данной модели приведены действия, которые связаны с анализом ее применимости для конкретного вида системы, а также обследование заказчика для определения потребностей пользователя для разработки плана создания прототипа.

Рисунок - 7**.** Модель эволюционного прототипирования

В модели есть две главные итерации разработки функционального прототипа, проектирования и реализации системы. Проверяется, удовлетворяет ли она всем функциональным и нефункциональным требованиям. Основной идеей этой модели

является моделирование отдельных функций системы в прототипе и постепенное эволюционная его доработка до выполнения всех заданных функциональных требований.

Итераций по получению промежуточных вариантов прототипа может быть несколько, в каждой из которых добавляется функция и повторно моделируется работа прототипа. И так до тех пор, пока не будут промоделированы все функции, заданные в требованиях к системе. Потом выполняется еще итерация - окончательное программирование для получения готовой системы.

Эта модель применяется для систем, в которых наиболее важными являются функциональные возможности, и которые необходимо быстро продемонстрировать на CASE-средствах.

Так как промежуточные прототипы системы *соответствуют реализации* некоторых функциональных требований, то их можно проверять и при сопровождении и эксплуатации, т.е. параллельно с процессом разработки очередных прототипов системы. При этом вспомогательные и организационные процессы могут выполняться параллельно с процессом разработки и накапливать сведения по данным количественных и качественных оценок на процессах разработки.

При этом учитываются такие факторы риска:

- реализация всех функций системы одновременно может привести к громоздкости;
- ограниченные человеческие ресурсы заняты разработкой в течение длительного времени.

Преимущества применения данной модели ЖЦ следующие:

- быстрая реализация некоторых функциональных возможностей системы и их апробирование;
- использование промежуточного продукта в следующем прототипе;
- выделение отдельных функциональных частей для реализации их в виде прототипа;
- возможность увеличения финансирования системы;
- обратная связь устанавливается с заказчиком для уточнения функциональных требований;
- упрощение внесения изменений в связи с заменой отдельной функции.

Модель развивается в направлении добавления нефункциональных требований к системе, связанных с защитой и безопасностью данных, несанкционированным доступом к ним и др.

1.3 Лекция №5, 6 (4 часа)

Тема: «Виды систем проектирования АСОИ»

1.3.1 Вопросы лекции:

1. Унифицированный процесс – управляемый вариантами использования.
2. Унифицированный процесс - ориентирован на архитектуру.
3. Унифицированный процесс - итеративный и инкрементный.
4. Жизненный цикл в унифицированном процессе.
5. Продукт унифицированного процесса.
6. Унифицированный процесс – методология разработки.

1.3.2 Краткое содержание вопросов:

1. Унифицированный процесс – управляемый вариантами использования.

Сегодня развитие программного обеспечения происходит в сторону увеличения и усложнения систем. Это связано отчасти с тем, что компьютеры с каждым годом становятся все мощнее, что побуждает пользователей ожидать от них все большего. Эта

тенденция также связана с возрастающим использованием Интернета для обмена всеми видами информации. Потребности в отношении более продвинутого программного обеспечения растут по мере того, как мы начинаем понимать с выходом каждого следующего выпуска, как еще можно улучшить этот продукт. Мы желаем иметь программное обеспечение, еще лучше приспособленное для наших нужд, а это, в свою очередь, приводит к усложнению программ.

Мы также желаем получить это программное обеспечение побыстрее. Время выхода на рынок - это другой важный стимул.

Сделать это, однако, нелегко. Наше желание получить мощные и сложные программы не сочетается с тем, как эти программы разрабатываются. Сегодня большинство людей разрабатывает программы, используя те же методы, что и 25 лет назад, что является серьезной проблемой. Если мы не улучшим наши методы, мы не сможем выполнить свою задачу по разработке так необходимого сегодня сложного программного обеспечения.

Проблема программного обеспечения сводится к затруднениям разработчиков, вынужденных преодолевать в ходе разработки больших программ множество преград. Общество разработчиков программного обеспечения нуждается в управляемом методе работы. Ему нужен процесс, который объединил бы множество аспектов разработки программ. Ему нужен общий подход, который:

- обеспечивал бы руководство деятельностью команды;
- управлял бы задачами отдельного разработчика и команды в целом;
- указывал бы, какие артефакты следует разработать;
- предоставлял бы критерии для отслеживания и измерения продуктов и функционирования проекта.

Унифицированный процесс разработки программного обеспечения - это решение проблемы программного обеспечения.

Прежде всего, Унифицированный процесс есть процесс разработки программного обеспечения. *Процесс разработки программного обеспечения - это сумма различных видов деятельности, необходимых для преобразования требований пользователей в программную систему*. Однако Унифицированный процесс - это больше, чем единичный процесс, это обобщенный каркас процесса, который может быть специализирован для широкого круга программных систем, различных областей применения, уровней компетенции и размеров проекта.

Унифицированный процесс компонентно-ориентирован. Это означает, что создаваемая программная система строится на основе программных компонентов, связанных хорошо определенными интерфейсами.

Для разработки чертежей программной системы Унифицированный процесс использует *Унифицированный язык моделирования*.

Однако действительно специфичные аспекты Унифицированного процесса заключаются в трех словосочетаниях - управляемый вариантами использования, архитектурно-ориентированный, итеративный и инкрементный. Это то, что делает Унифицированный процесс уникальным.

Программная система создается для обслуживания ее пользователей. Следовательно, для построения успешной системы мы должны знать, в чем нуждаются и чего хотят ее будущие пользователи.

Понятие *пользователь* относится не только к людям, работающим с системой, но и к другим системам. Таким образом, понятие *пользователь* относится к кому-либо или чему-либо, что взаимодействует с системой, которую мы разрабатываем. Пример взаимодействия - человек, использующий банкомат. Он вставляет в прорезь пластиковую карту, отвечает на вопрос, высвечиваемый машиной на экране, и получает деньги. В ответ на вставленную карту и ответы пользователя система осуществляет последовательность

действий, которые обеспечивают пользователю ощутимый и значимый для него результат, а именно получение наличных.

Взаимодействие такого рода называется вариантом использования. *Вариант использования - это часть функциональности системы, необходимая для получения пользователем значимого для него, ощутимого и измеримого результата.* Варианты использования обеспечивают функциональные требования. Сумма всех вариантов использования составляет модель вариантов использования, которая описывает полную функциональность системы. Однако варианты использования - это не только средство описания требований к системе. Они также направляют ее проектирование, реализацию и тестирование, то есть они направляют процесс разработки. Основываясь на модели вариантов использования, разработчики создают серию моделей проектирования и реализации, которые осуществляют варианты использования. Тестеры тестируют реализацию для того, чтобы гарантировать, что компоненты модели реализации правильно выполняют варианты использования. Таким образом, варианты использования не только инициируют процесс разработки, но и служат для связи отдельных его частей. *Управляемый вариантами использования* означает, что процесс разработки проходит серии рабочих процессов, порожденных вариантами использования.

2. Унифицированный процесс - ориентирован на архитектуру.

Понятие архитектуры программы включает в себя наиболее важные статические и динамические аспекты системы. Архитектура вырастает из требований к результату в том виде, как их понимают пользователи и другие заинтересованные лица. Эти требования отражаются в вариантах использования. Однако они также зависят от множества других факторов, таких как выбор платформы для работы программы (то есть компьютерной архитектуры, операционной системы, СУБД, сетевых протоколов), доступность готовых блоков многократного использования, соображения развертывания, унаследованные системы и нефункциональные требования. Архитектура - это представление всего проекта с выделением важных характеристик и затушевыванием деталей. Процесс помогает архитектору сконцентрироваться на правильных целях, таких, как понятность, легкость внесения изменений и возможность повторного использования.

Каждый продукт имеет функции и форму. Одно без другого не существует. В удачном продукте эти две стороны должны быть уравновешены. В этом примере функции соответствуют вариантам использования, а форма - архитектуре. Мы нуждаемся во взаимодействии между вариантами использования и архитектурой. Это вариант традиционной проблемы "курицы и яйца". Реально архитектура и варианты использования разрабатываются параллельно.

Таким образом, архитектор придает системе форму. Это означает, что форма, архитектура, должна быть спроектирована так, чтобы позволить системе развиваться не только в момент начальной разработки, но и в будущих поколениях. Чтобы найти такую форму, архитектор должен работать, полностью понимая ключевые функции, то есть ключевые варианты использования системы. Эти ключевые варианты использования составляют от 5 до 10 % всех вариантов использования и крайне важны, поскольку содержат функции ядра системы. Проще говоря, архитектор совершает следующие шаги:

- Создает грубый набросок архитектуры, начиная с той части архитектуры, которая не связана с вариантами использования.
- Далее архитектор работает с подмножеством выделенных вариантов использования, каждый из которых соответствует одной из ключевых функций разрабатываемой системы. Каждый из выбранных вариантов использования детально описывается и реа-лизуется в понятиях подсистем, классов и компонентов.

- После того как варианты использования описаны и полностью разработаны, большая часть архитектуры исследована, созданная архитектура, в свою очередь, будет базой для полной разработки других вариантов использования.

Этот процесс продолжается до тех пор, пока архитектура не будет признана стабильной.

3. Унифицированный процесс - итеративный и инкрементный.

Разработка коммерческих программных продуктов - это серьезное предприятие, которое может продолжаться от нескольких месяцев до года и более. Практично было бы разделить работу на небольшие куски или мини-проекты. Каждый мини-проект является итерацией, результатом которой будет приращение. Итерации относятся к шагам рабочих процессов, а приращение - к выполнению проекта. Для максимальной эффективности итерации должны быть управляемыми, то есть они должны выбираться и выполняться по плану. Поэтому их можно считать минипроектами.

Разработчики выбирают задачи, которые должны быть решены в ходе итерации, под воздействием двух факторов. Во-первых, в ходе итерации следует работать с группой вариантов использования, которая повышает применимость продукта в ходе дальнейшей разработки. Во-вторых, в ходе итерации следует заниматься наиболее серьезными рисками. Последовательные итерации используют артефакты разработки в том виде, в котором они остались при окончании предыдущей итерации. Это минипроекты, поскольку для выбранных вариантов использования проводится последовательная разработка - анализ, проектирование, реализация и тестирование, и в результате варианты использования, которые разрабатывались в ходе итерации, представляются в виде исполняемого кода.

На каждой итерации разработчики определяют и описывают уместные варианты использования, создают проект, использующий выбранную архитектуру в качестве направляющей, реализуют проект в компоненты и проверяют соответствие компонентов вариантам использования. Если итерация достигла своей цели, процесс разработки переходит на следующую итерацию. Если итерация не выполнила своей задачи, разработчики должны пересмотреть свои решения и попробовать другой подход.

Для получения максимальной экономии команда, работающая над проектом, должна выбирать только те итерации, которые требуются для выполнения цели проекта. Для этого следует расположить итерации в логической последовательности. Разумеется, до определенного предела; так, незамеченные ранее проблемы приведут к добавлению итераций или изменению их последовательности, и процесс разработки потребует больших усилий и времени. Минимизация незамеченных проблем является одной из целей снижения рисков.

Управляемый итеративный процесс имеет множество преимуществ.

- Управляемая итерация ограничивает финансовые риски затратами на одно приращение. Если разработчикам нужно повторить итерацию, организация теряет только усилия, затраченные на одну итерацию, а не стоимость всего продукта.
- Управляемая итерация снижает риск непоставки продукта на рынок в запланированные сроки. При раннем обнаружении соответствующего риска время, которое тратится на его нейтрализацию, вносится в план на ранних стадиях, когда сотрудники менее загружены, чем в конце планового периода.
- Управляемая итерация ускоряет темпы процесса разработки в целом, поскольку для более эффективной работы разработчиков и быстрого получения ими хороших результатов короткий и четкий план предпочтительнее длинного и вечно сдвигающегося.
- Управляемая итерация признает тот факт, что желания пользователей и связанные с ними требования не могут быть определены в начале разработки. Они обычно уточняются в последовательных итерациях.

Эти концепции - управляемая вариантами использования, архитектурно-ориентированная и итеративная и инкрементная разработка - одинаково важны. Архитектура предоставляет нам структуру, направляющую нашу работу в итерациях, в каждой из которых варианты использования определяют цели и направляют нашу работу.

1. Жизненный цикл в унифицированном процессе.

Унифицированный процесс циклически повторяется. Эта последовательность повторений Унифицированного процесса представляет собой жизненный цикл системы. Каждый цикл завершается поставкой выпуска продукта заказчикам.

Каждый цикл состоит из четырех фаз - анализа и планирования требований, проектирования, построения и внедрения. Каждая фаза, как будет рассмотрено ниже, далее подразделяется на итерации.

Каждый цикл осуществляется в течение некоторого времени. Это время, в свою очередь, делится на четыре фазы: фазу анализа и планирования требований, фазу проектирования, фазу построения и фазу внедрения. Внутри каждой фазы руководители или разработчики могут потерпеть неудачу в работе - но только на данной итерации и в связанном с ней приращении. Каждая фаза заканчивается вехой. Мы определяем каждую веху по наличию определенного набора артефактов, например, некая модель документа должна быть приведена в предписанное состояние.

Вехи служат многим целям. Наиболее важная из них - дать руководителям возможность принять некоторые критические решения перед тем, как работа перейдет на следующую фазу. Вехи также дают руководству и самим разработчикам возможность отслеживать прогресс в работе при проходах этих четырех ключевых точек.

В ходе фазы анализа и планирования требований хорошая идея превращается в концепцию готового продукта и создается бизнесплан разработки этого продукта. В частности, на этой фазе должны быть получены ответы на вопросы:

- Что система должна делать в первую очередь для ее основных пользователей?
- Как должна выглядеть архитектура системы?
- Каков план и во что обойдется разработка продукта?

Упрощенная модель вариантов использования, содержащая наиболее критичные варианты использования, дает ответ на первый вопрос. На этом этапе создается пробный вариант архитектуры. Обычно он представляет собой набросок, содержащий наиболее важные подсистемы. На этой фазе выявляются и расставляются по приоритетности наиболее важные риски, детально планируется фаза проектирования и грубо оценивается весь проект.

В ходе фазы проектирования детально описываются большинство вариантов использования и разрабатывается архитектура системы.

Архитектура определяется в виде представлений всех моделей системы, которые в сумме представляют систему целиком. Это означает, что существуют архитектурные представления модели вариантов использования, модели анализа, модели проектирования, модели реализации и модели развертывания. Представление модели реализации включает в себя компоненты для доказательства того, что архитектура выполнима. В ходе этой фазы определяются наиболее критичные варианты использования. Результатом выполнения этой фазы является базовый уровень архитектуры.

В конце фазы проектирования менеджер проекта занимается планированием действий и подсчетом ресурсов, необходимых для завершения проекта. Ключевым вопросом в этот момент будет следующий: достаточно ли проработаны варианты использования, архитектура и план и взяты ли риски под контроль настолько, чтобы можно было давать контрактные обязательства выполнить всю работу по разработке?

В ходе фазы построения происходит создание продукта - к скелету (архитектуре) добавляются мышцы (законченные программы). На этой фазе базовый уровень

архитектуры разрастается до полной развитой системы. Концепции развиваются до продукта, готового к передаче пользователям. В ходе фазы объем требуемых ресурсов вырастает. Архитектура системы стабильна, однако, поскольку разработчики могут найти лучшие способы структурирования системы, от них могут исходить предложения о внесении в архитектуру системы небольших изменений. В конце этой фазы продукт включает в себя все варианты использования, которые руководство и заказчик договорились включить в текущий выпуск. Правда, они могут содержать ошибки. Большинство дефектов будут обнаружены и исправлены в ходе фазы внедрения. Ключевой вопрос окончания фазы: удовлетворяет ли продукт требованиям пользователей настолько, что некоторым заказчикам можно делать предварительную поставку?

Фаза внедрения охватывает период, в ходе которого продукт существует в виде бета-выпуска или бета-версии. Небольшое число квалифицированных пользователей, работая с бета-выпуском продукта, сообщает об обнаруженных дефектах и недостатках. После этого разработчики исправляют обнаруженные ошибки и вносят некоторые из предложенных улучшений в главный выпуск, подготавливаемый для широкого распространения. Фаза внедрения включает в себя такие действия, как производство тиража, тренинг сотрудников заказчика, организацию поддержки по горячей линии и исправление дефектов, обнаруженных после поставки.

Итогом проекта по разработке программного обеспечения является продукт, в создании которого принимает участие множество различных людей.

2. Продукт унифицированного процесса.

Результатом каждого цикла является новый выпуск системы, а каждый выпуск - это продукт, готовый к поставке. Он включает в себя тело - исходный код, воплощенный в компоненты, которые могут быть откомпилированы и выполнены, плюс руководство и дополнительные компоненты поставки. Однако готовый продукт должен также быть приспособлен для нужд не только пользователей, а всех заинтересованных лиц. Программному продукту следовало бы представлять собой нечто большее, чем исполняемый машинный код.

Окончательный продукт включает в себя требования, варианты использования, нефункциональные требования и варианты тестирования. Он включает архитектуру и визуальные модели - артефакты, смоделированные на Унифицированном языке моделирования. Эти средства позволяют заинтересованным лицам использовать систему и модифицировать ее от поколения к поколению.

Даже если исполняемые компоненты с точки зрения пользователей являются важнейшим артефактом, их одних недостаточно. Системное окружение меняется. Операционные системы, СУБД и сами компьютеры прогрессируют. По мере того как цель понимается все лучше, изменяются и требования. В конце концов, разработчики вынуждены начинать новый цикл, а руководство вынуждено их финансировать. Для того чтобы эффективно выполнять новый цикл, разработчикам необходимо иметь полное представление о программном продукте:

- модель вариантов использования со всеми вариантами использования и их связями с пользователями;
- модель анализа, которая имеет две цели - уточнить детали вариантов использования и создать первичное распределение поведения системы по набору объектов, предоставляющих различные варианты поведения;
- модель проектирования, которая определяет статическую структуру системы, такую как подсистемы, классы и интерфейсы, и варианты использования, реализованные в виде кооперации между подсистемами, классами и интерфейсами;
- модель реализации, которая включает в себя компоненты (представленные исходным кодом) и раскладку классов по компонентам;

- модель развертывания, которая определяет физические компьютеры - узлы сети и раскладку компонентов по этим узлам;
- модель тестирования, которая описывает варианты тестов для проверки вариантов использования;
- и, разумеется, представление архитектуры.

Все эти модели связаны. Вместе они полностью описывают систему. Элементы одной модели имеют трассировочные зависимости вперед и назад, организуемые с помощью связей с другими моделями. Например, вариант использования может быть оттрассирован на соответствующую реализацию варианта использования в модели проектирования и вариант тестирования в модели тестирования.

3. Унифицированный процесс – методология разработки.

Унифицированный процесс как процесс разработки программного обеспечения представляет собой методологию, содержащую детальное описание работ по созданию и внедрению ПО (авторы Айвар Якобсон, Грэди Буч и Джеймс Рамбо). Она отвечает "на вопросы *когда*, *как, кто, что* и *с помощью чего* реализуется проект", а именно содержит описание:

- технологических процессов (*когда*) – последовательности видов деятельности (работ), дающих ощутимый результат. Технологический процесс, как правило, представляется в виде диаграммы, отображающей состав работ и их последовательность на той или иной стадии разработки ПО;
- видов деятельности (*как*) – работ, осуществляемых исполнителями;
- исполнителей (*кто*) – заинтересованных в реализации проекта отдельных лиц или групп. Исполнитель характеризуется строго определенным поведением и обязанностями (ролью). Поведение выражается через виды деятельности, осуществляемые исполнителем, а обязанности – через результаты, получаемые в процессе выполнения работ. В процессе реализации проекта один и тот же человек может выступать в разных ролях;
- артефактов (*что*) – информации, создаваемой, изменяемой или используемой исполнителями в проекте. Другими словами, артефакт – это не только то, что создается в результате деятельности (***технические артефакты*** – модели системы, исходные коды программ, готовый программный продукт, документация к нему и т. д.), но и то, что направляет эту деятельность (***артефакты управления*** –календарный план, техническое задание, инструкции и т. д.);
- используемых утилит (*с помощью чего*) – программных продуктов, рекомендуемых при выполнении работ.

Унифицированный процесс придерживается спиральной модели (стратегии) жизненного цикла ПО, предложенной Барри Боэмом.

Каждый виток характеризуется приращением (инкрементом) функциональности системы и одинаковым набором технологических процессов и стадий. В рамках одной стадии также используется идея спиральной разработки. Перед началом выполнения каждой стадии планируется количество итераций, каждая из которых характеризуется некоторым приращением результатов. В рамках одной итерации выполняются основные процессы, начиная от формирования требований и заканчивая внедрением.

В Унифицированном процессе принято временное разбиение жизненного цикла на четыре стадии: начало, уточнение, конструирование и переход. Каждая стадия должна завершаться достижением конкретного результата (созданием артефактов), используемого далее в качестве управления последующими работами или завершающего реализацию проекта.

Все виды деятельности направлены на создание артефактов, самым главным и ценным из которых является разработанная информационная система. С точки зрения разработчиков не менее ценными артефактами являются разработанные модели системы,

так как они, с одной стороны, фиксируют результаты одних работ, а с другой – выступают в качестве управляющей и направляющей информации для других. В Унифицированном процессе модели, как правило, соответствуют основным технологическим процессам. Каждая модель представляет собой набор взаимосвязанных диаграмм UML и документов. Краткая характеристика моделей дана в следующей таблице.

Таблица 1. Краткая характеристика моделей

Процесс	Модель	Назначение
Формирование требований	Модель вариантов использования	Отображает существенные функциональные требования к системе в форме, удобной для всех заинтересованных лиц. Под существенными требованиями понимаются требования, реализация которых принесет пользователям ощутимый и значимый результат. Наименее формальная, но управляет всем процессом разработки
Анализ требований	Модель анализа	Детализирует варианты использования с точки зрения организации внутренней архитектуры системы, а именно: состава основных сущностей (классов анализа) и взаимодействия между ними. Класс анализа – укрупненный класс (сущность), который в дальнейшем будет разбит на составляющие
Проектирование	Модель проектирования	Содержит полное детализированное описание внутренней архитектуры и алгоритмов работы системы. Применяется внутри организации, разрабатывающей систему
Реализация	Модель реализации	Содержит описание исполняемой системы: компонентов (исходных текстов программ, исполняемых модулей, таблиц БД и т. д.) и схемы развертывания системы
Тестирование	Модель тестирования	Предназначена для проверки соответствия полученного ПО требованиям

Модели описывают проектируемую систему с различных точек зрения и на разном уровне абстракции. При этом некоторые элементы (например, диаграммы или классы) могут одновременно входить в разные модели. Более того, один и тот же элемент может входить в две и более моделей с разной степенью детализации.

Модели могут быть вложены друг в друга. Вложенность моделей изображается двумя способами.

В Унифицированном процессе набор получаемых артефактов, как и технологический процесс, также может отображаться в виде диаграммы. На следующем рисунке показан пример диаграммы артефактов для процесса "Управление проектом".

Качественное и своевременное выполнение проекта невозможно без применения средств автоматизации деятельности – утилит. К утилитам относятся различные инструментальные средства, поддерживающие жизненный цикл ПО. В качестве примера системного подхода к разработке информационных систем можно привести продукты компании IBM Rational, лидера в разработке и сопровождении средств, поддерживающих создание объектно-ориентированных систем:

- управление требованиями – IBM RationalRequisitePro;
- визуальное моделирование и генерация объектного кода – IBM RationalRose, IBM Rational XDE;
- разработку – IBM Rational RapidDeveloper;

- конфигурационное управление – IBM Rational ClearCase;
- управление изменениями – IBM RationalClearQuest;
- автоматизированное документирование – IBM RationalSoDA;
- автоматизированное тестирование – IBM Rational TeamTest, IBM Rational TestFactory, IBM Rational Robot, IBM Rational PurifyPlus, IBM Rational SiteCheckи IBM Rational SiteLoad.

Для эффективного применения этих идей необходимо, чтобы они образовывали единый многоплановый процесс, поддерживающий циклы, фазы, рабочие процессы, снижение рисков, контроль качества, управление проектом и конфигурацией. Унифицированный процесс создает каркас, объединяющий все аспекты. Такой комплекс знаний, называют методологией разработки.

1.4 Лекция №7, 8, 9 (6 часов)

Тема: «Типы диаграмм в языке UML**»**

1.4.1 Вопросы лекции:

1. Классификация диаграмм, принятые обозначения.
2. Изображение ассоциаций на диаграммах классов.
3. Иерархии классов.
4. CRC-карточки.
5. Диаграмма классов.
6. Статические (static) и динамические классы.
7. Диаграммы объектов.
8. Диаграммы прецедентов.
9. Диаграмма состояний (конечных автоматов).
10. Диаграммы последовательностей.
11. Диаграммы коммуникации (взаимодействия).
12. Зачем так много различных диаграмм?
13. Диаграммы видов деятельности.
14. Диаграммы компонентов.
15. Диаграммы развертывания.
16. Диаграммы пакетов.
17. Временная диаграмма.

1.4.2 Краткое содержание вопросов:

1. Классификация диаграмм, принятые обозначения.

UML 2 описывает 13 официальных типов диаграмм, перечисленных в табл. 1.1, классификация которых приведена на рис. 1.2. Хотя эти виды диаграмм отражают различные подходы многих специалистов к UML и способ организации моей книги, авторы UML не считают диа граммы центральной составляющей языка. Поэтому диаграммы опре делены не очень строго. Часто вполне допустимо присутствие элементов диаграммы одного типа в другой диаграмме. Стандарт UML указывает, что определенные элементы обычно рисуются в диаграммах соответствующего типа, но это не догма.

Допустимый UML – это язык, определенный в соответствии со спецификацией. Однако на практике ответ несколько сложнее.

- Структурные диаграммы:
  - диаграммы классов (class diagrams) предназначены для моделирования структуры объектно-ориентированных приложений - классов, их атрибутов и заголовков методов, наследования, а также связей классов друг с другом;

- диаграммы компонент (component diagrams) используются при моделировании компонентной структуры распределенных приложений; внутри каждая компонента может быть реализована с помощью множества классов;
- диаграммы объектов (object diagrams) применяются для моделирования фрагментов работающей системы, отображая реально существующие в runtime экземпляры классов и значения их атрибутов;
- диаграммы композитных структур (composite structure diagrams) используются для моделирования составных структурных элементов моделей - кооперации, композитных компонент и т.д.;
- диаграммы развертывания (deployment diagrams) предназначены для моделирования аппаратной части системы, с которой ПО непосредственно связано (размещено или взаимодействует);
- диаграммы пакетов (package diagrams) служат для разбиения объемных моделей на составные части, а также (традиционно) для группировки классов моделируемого ПО, когда их слишком много.

• Поведенческие диаграммы:

- диаграммы активностей (activity diagrams) используются для спецификации бизнес-процессов, которые должно автоматизировать разрабатываемое ПО, а также для задания сложных алгоритмов;
- диаграммы случаев использования (use case diagrams) предназначены для "вытягивания" требований из пользователей, заказчика и экспертов предметной области;
- диаграммы конечных автоматов (state machine diagrams) применяются для задания поведения реактивных систем;
- диаграммы взаимодействий (interaction diagrams):
  - диаграммы последовательностей (sequence diagrams) используются для моделирования временных аспектов внутренних и внешних протоколов ПО;
  - диаграммы схем взаимодействия (interaction overview diagrams) служат для организации иерархии диаграмм последовательностей;
  - диаграммы коммуникаций (communication diagrams) являются аналогом диаграмм последовательностей, но по-другому изображаются (в привычной, графовой, манере);
  - временные диаграммы (timing diagrams) являются разновидностью диаграмм последовательностей и позволяют в наглядной форме показывать внутреннюю динамику взаимодействия некоторого набора компонент системы.

Существенным в вопросе является то, на каких правилах базируется UML: описательных или предписывающих. Язык с **предписывающими правилами** (prescriptiverules) управляется официальной основой, которая устанавливает, что является, а что не является допустимым языком, и какое значение вкладывается в понятие высказывания языка. Язык с **описательными правилами** (descriptiverules) – это язык, правила которого распознаются по тому, как люди применяют его на практике. Языки программирования в основном имеют предписываю щие правила, установленные комитетом по стандартам или основными поставщиками, тогда как естественные языки, такие как английский, в основном имеют описательные правила, смысл которых устанавливается по соглашению.

UML – точный язык, поэтому можно было бы ожидать, что он основан на предписывающих правилах. Но UML часто рассматривают как программный эквивалент чертежей из других инженерных дисциплин, а эти чертежи основаны не на предписывающих нотациях. Никакой комитет не говорит, какие символы являются законными встроитель ной технической документации; эти нотации были приняты по соглашению, как и в естественном языке. Стандарты сами по себе еще ничего не решают, поскольку те, кто работает в этой области, не смогут следовать всему, что указывают стандарты; это то же самое, что спрашивать французов о французской академии наук. К тому же язык UML настолько сложен, что стандарты часто можно трактовать по разному.

Даже ведущие специалисты по UML, которые рецензировали эту книгу, не согласились бы интерпретировать стандарты.

Этот вопрос важен и для меня, пишущего эту книгу, и для вас, применяющих язык UML. Если вы хотите понять диаграммы UML, важно уяснить, что понимание стандартов – это еще не вся картина. Люди принимают соглашения и в индустрии в целом, и в каких-то конкретных проектах. Поэтому, хотя стандарт UML и может быть первичным источником информации по UML, он не должен быть единственным.

Моя позиция состоит в том, что для большинства людей UML имеет описательные правила. Стандарт UML оказывает наибольшее влияние на содержание UML, но это делает не только он. Я думаю, что особенно верным это станет для UML 2, который вводит некоторые соглашения по обозначениям, конфликтующие или с определениями UML 1, или с использованием по соглашению, а также еще больше усложняет язык. Поэтому в данной книге я стараюсь обобщить UML так, как я его вижу: и стандарты, и применение по соглашению. Когда мне приходится указывать на некоторое отличие в этой книге, я буду употреблять термин **применение по соглашению** (conventionaluse), чтобы обо значить то, чего нет в стандарте, но, как я думаю, широко применяется. В случае если что то соответствует стандарту, я буду употреблять термин **стандартный** (standard) или **нормативный** (normative). (Нормативный – это термин, посредством которого люди обозначают утверждение, которое вы должны подтвердить, чтобы оно соответствовало стандарту.Поэтому выражение ненормативный UML – это своеобразный способ сказать, что нечто совершенно неприемлемо с точки зрения стандарта UML.)

Рассматривая диаграмму UML, необходимо помнить, что основной принцип UML заключается в том, что любая информация на конкрет ной диаграмме может быть подавлена. Это подавление может носить глобальный характер – скрыть все атрибуты – или локальный – не показывать какие-нибудь конкретные классы. Поэтому по диаграмме вы никогда не можете судить о чем-нибудь по его отсутствию. Даже если метамодель UML имеет поведение по умолчанию, например [1] для атрибутов, когда вы не видите эту информацию на диаграмме, это может быть обусловлено либо поведением по умолчанию, либо тем, что она просто подавлена.

Говоря это, следует упомянуть, что существуют основные соглашения, например о том, что многозначные свойства должны быть множествами.

Не надо слишком зацикливаться на допустимом UML, если вы занимаетесь эскизами или моделями. Важнее составить хороший проект системы, и я предпочитаю иметь хороший дизайн в недопустимом UML, чем допустимый UML, но плохой дизайн. Очевидно, хороший и допустимый предпочтительнее, но лучше направить свою энергию на разработку хорошего проекта, чем беспокоиться о секретах UML. (Конечно, в случае применения UML в качестве языка программирования необходимо соблюдать стандарты, иначе программа будет работать неправильно!)

Смысл UML

Одним из затруднений в UML является то, что хотя спецификация по дробно описывает определение правильно сформированного UML, но этого недостаточно, чтобы определить значение UML вне сферы изысканного выражения «метамодель UML». Не существует формальных описаний того, как UML отображается на конкретные языки программирования. Вы не можете посмотреть на диаграмму UML и *точно* сказать, как будет выглядеть соответствующий код. Однако у вас может быть *приблизительное представление* о виде программы. На практике этого достаточно. Команды разработчиков часто формируют собст венные локальные соглашения, и, чтобы их использовать, вам придет ся с ними познакомиться.

2. Изображение ассоциаций на диаграммах классов.

Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.

Представление классов

Класс – это основной строительный блок ПС. Это понятие присутствует и в ОО языках программирования, то есть между классами UML и программными классами есть соответствие, являющееся основой для автоматической генерации программных кодов или для выполнения реинжиниринга. Каждый класс имеет название, атрибуты и операции. Класс на диаграмме показывается в виде прямоугольника, разделенного на 3 области. В верхней содержится название класса, в средней – описание атрибутов (свойств), в нижней – названия операций – услуг, предоставляемых объектами этого класса.

Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет. При реализации объекта в программном коде для атрибутов будет выделена память, необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть сколь угодно много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы.

Для каждого атрибута класса можно задать видимость (visibility). Эта характеристика показывает, доступен ли атрибут для других классов. В UML определены следующие уровни видимости атрибутов:

- Открытый (public) – атрибут виден для любого другого класса (объекта);
- Защищенный (protected) – атрибут виден для потомков данного класса;
- Закрытый (private) – атрибут не виден внешними классами (объектами) и может использоваться только объектом, его содержащим.

Последнее значение позволяет реализовать свойство инкапсуляции данных. Например, объявив все атрибуты класса закрытыми, можно полностью скрыть от внешнего мира его данные, гарантируя отсутствие несанкционированного доступа к ним. Это позволяет сократить число ошибок в программе. При этом любые изменения в составе атрибутов класса никак не скажутся на остальной части ПС.

Класс содержит объявления **операций**, представляющих собой определения запросов, которые должны выполнять объекты данного класса. Каждая операция имеет **сигнатуру**, содержащую имя операции, тип возвращаемого значения и список параметров, который может быть пустым. Реализация операции в виде процедуры – это метод, принадлежащий классу. Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в ПС.

Отношения

На диаграммах классов обычно показываются ассоциации и обобщения (см. предыдущую статью).

Каждая **ассоциация** несет информацию о связях между объектами внутри ПС. Наиболее часто используются бинарные ассоциации, связывающие два класса. Ассоциация может иметь название, которое должно выражать суть отображаемой связи (см. рис. 2). Помимо названия, ассоциация может иметь такую характеристику, как **множественность**. Она показывает, сколько объектов каждого класса может участвовать в ассоциации. Множественность указывается у каждого конца ассоциации

(полюса) и задается конкретным числом или диапазоном чисел. Множественность, указанная в виде звездочки, предполагает любое количество (в том числе, и ноль).

Ассоциация сама может обладать свойствами класса, то есть, иметь атрибуты и операции. В этом случае она называется класс-ассоциацией и может рассматриваться как класс, у которого помимо явно указанных атрибутов и операций есть ссылки на оба связываемых ею класса. В примере на рис.2 ассоциация «включает» по существу есть класс-ассоциация, укоторой есть атрибут «Количество», показывающий, сколько единиц каждого товара входит в набор.

Обобщение на диаграммах классов используется, чтобы показать связь между классом-родителем и классом-потомком. Оно вводится на диаграмму, когда возникает разновидность какого-либо класса (например, при развитии ПС – см. рис.4), а также в тех случаях, когда в системе обнаруживаются несколько классов, обладающих сходным поведением.

Как уже говорилось ранее, UML позволяет строить модели с различным уровнем детализации.

Стереотипы классов

При создании диаграмм классов часто пользуются понятием «стереотип». В дальнейшем речь пойдет о стереотипах классов. **Стереотип** класса – это элемент расширения словаря UML, который обозначает отличительные особенности в использовании класса. Стереотип имеет название, которое задается в виде текстовой строки. При изображении класса на диаграмме стереотип показывается в верхней части класса в двойных угловых скобках. Есть четыре стандартных стереотипа классов, для которых предусмотрены специальные графические изображения (см. рис.5).

Стереотип используется для обозначения классов-сущностей (классов данных), стереотип описывает пограничные классы, которые являются посредниками между ПС и внешними по отношению к ней сущностями – актерами, обозначаемыми стереотипом <>. Наконец, стереотип описывает классы и объекты, которые управляют взаимодействиями. Применение стереотипов позволяет, в частности, изменить вид диаграмм классов.

Применение диаграмм классов

Диаграммы классов создаются при логическом моделировании ПС и служат для следующих целей:

- Для моделирования данных. Анализ предметной области позволяет выявить основные характерные для нее сущности и связи между ними. Это удобно моделируется с помощью диаграмм классов. Эти диаграммы являются основой для построения концептуальной схемы базы данных.
- Для представления архитектуры ПС. Можно выделить архитектурно значимые классы и показать их на диаграммах, описывающих архитектуру ПС.
- Для моделирования навигации экранов. На таких диаграммах показываются пограничные классы и их логическая взаимосвязь. Информационные поля моделируются как атрибуты классов, а управляющие кнопки – как операции и отношения.
- Для моделирования логики программных компонент (будет описано в последующих статьях).
- Для моделирования логики обработки данных.

3. Иерархии классов.

Самым распространенным видом отношения зависимости является соединение между классами, когда один класс использует другой в качестве параметра операции.

Для моделирования такого отношения изобразите зависимость, направленную от класса с операцией к классу, используемому в качестве ее параметра.

Одиночное наследование

Моделируя словарь системы, вам часто придется работать с классами, похожими на другие по структуре и поведению. В принципе их можно моделировать как различные, независимые друг от друга абстракции. Но лучше выделить одинаковые свойства и сформировать на их основе общие классы, которым наследуют специализированные.

Моделирование отношений наследования осуществляется в таком порядке:

1. Найдите атрибуты, операции и обязанности, общие для двух или более классов из данной совокупности
2. Вынесите эти элементы в некоторый общий класс (если надо, создайте новый, но следите, чтобы уровней не оказалось слишком много).
3. Отметьте в модели, что более специализированные классы наследуют более общим, включив отношение обобщения, направленное от каждого потомка к его родителю.

Структурные отношения

Отношения зависимости и обобщения применяются при моделировании классов, которые находятся на разных уровнях абстракции или имеют различную значимость. Что касается отношения зависимости, один класс зависит от другого, но тот может ничего не "знать" о наличии первого. Когда речь идет об отношении обобщения, класс-потомок наследует своему родителю, но сам родитель о нем не осведомлен. Другими словами, отношения зависимости и обобщения являются односторонними.

Ассоциации предполагают участие равноправных классов. Если между двумя классами установлена ассоциация, то каждый из них каким-то образом зависит от другого, и навигацию можно осуществлять в обоих направлениях. В то время как зависимость - это отношение использования, а обобщение - отношение "является", ассоциации определяют структурный путь, обусловливающий взаимодействие объектов данных классов. По умолчанию ассоциации являются двунаправленными, но вы можете оставить только одно направление.

Моделирование структурных отношений производится следующим образом:

1. Определите ассоциацию для каждой пары классов, между объектами которых надо будет осуществлять навигацию. Это взгляд на ассоциации с точки зрения данных.
2. Если объекты одного класса должны будут взаимодействовать с объектами другого иначе, чем в качестве параметров операции, следует определить между этими классами ассоциацию. Это взгляд на ассоциации с точки зрения поведения.
3. Для каждой из определенных ассоциаций задайте кратность (особенно если она не равна *, то есть значению по умолчанию) и имена ролей (особенно если это помогает объяснить модель).
4. Если один из классов ассоциации структурно или организационно представляет собой целое в отношении классов на другом конце ассоциации, выглядящих как его части, пометьте такую ассоциацию как агрегирование.

Как узнать, когда объекты данного класса должны взаимодействовать с объектами другого класса? Для этого рекомендуется воспользоваться CRC-карточка-ми и анализом прецедентов (см. главу 16). Эти методы очень помогают при рассмотрении структурных и поведенческих вариантов функционирования системы. Если в результате обнаружится взаимодействие между классами, специфицируйте ассоциацию.

4. CRC-карточки.

CRC-карточки. CRC обозначает Class-Responsibilities-Collaborators (Класс/Ответственности/Участники). Это эффективный способ анализа сценариев. Карты CRC впервые предложили Бек и Каннингхэм для обучения объектно-ориентированному программированию.

Карта представляет собой таблицу с заголовком и двумя столбцами. На карточках разработчик пишет сверху – название класса, базового или суперкласса, производных классов, снизу в левой половине – за что он отвечает, а в правой половине – с кем он сотрудничает. Проходя по сценарию, разработчик заводит по карточке на каждый обнаруженный класс и дописывает в нее новые пункты. При этом каждый раз анализируется, что из этого получается, и "выделяется излишек ответственности" в новый класс или переносятся ответственности с одного большого класса на несколько более детальных классов, или, возможно, передается часть обязанностей другому классу.

Т.е. это вполне нормальная ситуация, когда разработчику необходимо переделывать ОО словарь и CRC-карты несколько раз, что говорит о важности этапа анализа.

Карточки можно раскладывать так, чтобы представить формы сотрудничества объектов. С точки зрения динамики сценария, их расположение может показать поток сообщений между объектами, с точки зрения статики они представляют иерархии классов.

5. Диаграмма классов.

Диаграмма классов описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами. В UML термин **функциональность** (feature) применяется в качестве основного термина, описывающего и свойства, и операции класса.

Свойства представляют структурную функциональность класса. В первом приближении можно рассматривать свойства как поля класса. Как мы увидим позднее, в действительности это не так просто, но вполне приемлемо для начала.

Свойства представляют единое понятие, воплощающееся в двух совершенно различных сущностях: в атрибутах и в ассоциациях. Хотя на диаграмме они выглядят совершенно по разному, в действительности это одно и то же.

Атрибут описывает свойство в виде строки текста внутри прямоугольника класса. Полная форма атрибута:

видимость имя: тип кратность = значение по умолчанию {строка свойств}

Например:

имя: String [1] = "Без имени" {readOnly}

Обязательно только имя.

•Метка **видимость** обозначает, относится ли атрибут к открытым (+) (public) или к закрытым ( ) (private).

•**Имя** атрибута – способ ссылки класса на атрибут – приблизительно соответствует имени поля в языке программирования.

•**Тип** атрибута накладывает ограничение на вид объекта, который может быть размещен в атрибуте. Можно считать его аналогом типа поля в языке программирования.

•**Кратность.**

•**Значение по умолчанию** представляет собой значение для вновь создаваемых объектов, если атрибут не определен в процессе создания.

•Элемент **{строка свойств}** позволяет указывать дополнительные свойства атрибута. В примере он равен**{readOnly}**, то есть клиенты не могут изменять атрибут. Если он пропущен, то, как правило, атрибут можно модифицировать. Остальные строки свойств будут описаны позже.

Ассоциации

Другая ипостась свойства – это ассоциация. Значительная часть ин формации, которую можно указать в атрибуте, появляется в ассоциации.

Order

+dateReceived: Date [0..1]

+isPrepaid: Boolean [1]
+lineItems: OrderLine [*] {ordered}

Ассоциация – это непрерывная линия между двумя классами, направленная от исходного класса к целевому классу. Имя свойства (вместе с кратностью) располагается на целевом конце ассоциации. Целевой конец ассоциации указывает на класс, который является типом свойства. Большая часть информации в обоих представлениях одинакова, но некоторые элементы отличаются друг от друга. В частности, ассоциация может показывать кратность на обоих концах линии.

Естественно, возникает вопрос: «Когда следует выбирать то или иное представление?». Как правило, я стараюсь обозначать при помощи атрибутов небольшие элементы, такие как даты или логические значения, – главным образом, типы значений, – а ассоциации для более значимых классов, таких как клиенты или заказы. Я также предпочитаю использовать прямоугольники классов для наиболее значимых классов диаграммы, а ассоциации и атрибуты для менее важных элементов этой диаграммы.

Кратность свойства обозначает количество объектов, которые могут заполнять данное свойство. Чаще всего встречаются следующие кратности:

•1 (Заказ может представить только один клиент.)

•0..1 (Корпоративный клиент может иметь, а может и не иметь единственного торгового представителя.)

•* (Клиент не обязан размещать заказ, и количество заказов не ограничено.Он может разместить ноль или более заказов.)

В большинстве случаев кратности определяются своими нижней и верхней границами, например 2..4 для игроков в канасту. Нижняя граница может быть нулем или положительным числом, верхняя граница представляет собой положительное число или * (без ограничений). Если нижняя и верхняя границы совпадают, то можно указать одно число; поэтому 1 эквивалентно 1..1. Поскольку это общий случай, * является сокращением 0..*.

При рассмотрении атрибутов вам могут встретиться термины, имеющие отношение к кратности.

•**Optional** (необязательный) предполагает нулевую нижнюю границу.

•**Mandatory** (обязательный) подразумевает, что нижняя граница равна или больше 1.

•**Singlevalued** (однозначный) – для такого атрибута верхняя грани ца равна 1.

•**Multivalued** (многозначный) имеет в виду, что верхняя граница больше 1; обычно *.

Если свойство может иметь несколько значений, я предпочитаю употреблять множественную форму его имени. По умолчанию элементы с множественной кратностью образуют множество, поэтому если вы просите клиента разместить заказы, то они приходят не в произвольном порядке. Если порядок заказов в ассоциации имеет значение, то в конце ассоциации необходимо добавить **{ordered}**. Если вы хотите разрешить повторы, то добавьте**{nonunique}**. (Если желательно явным образом показать значение по умолчанию, то можно использовать**{unordered}** и**{unique}**.) Встречаются также имена для unordered, nonunique, ориентированные на коллекции, такие как**{bag}**.

UML 1 допускал дискретные кратности, например 2,4 (означающую 2 или 4, как в случае автомобилей, до того как появились минивэны). Дискретные кратности не были широко распространены, и в UML 2 их уже нет.

Кратность атрибута по умолчанию равна. Хотя это и верно для метамодели, нельзя предполагать, что если значение кратности для атрибута на диаграмме опущено, то оно равно, поскольку информация о кратности на диаграмме может отсутствовать. Поэтому я пред почитаю указывать кратность явным образом, если эта информация важна.

Программная интерпретация свойств

Как и для других элементов UML, интерпретировать свойства в про грамме можно по разному. Наиболее распространенным представлением является поле или свойство языка программирования. Так, класс **OrderLine** (Строка заказа), показанный на рис. 3.1, мог бы быть представлен в Java следующим образом:

public class OrderLine...

privateint quantity; private Money price; private Order order; private Product product

В языке, подобном C#, который допускает свойства, это могло бы вы глядеть так:

public class OrderLine ...

publicint Quantity; public Money Price; public Order Order; public Product Product;

Обратите внимание, что атрибут обычно соответствует открытым (public) свойствам в языке, поддерживающем свойства, но соответствует закрытым (private) полям в языке, в котором такой поддержки нет. В языке без свойств с полями можно общаться посредством методов доступа (получение и установка). У атрибута только для чтения не бу дет метода установки (в случае полей) или операции установки (в случае свойства). Учтите, что если свойству не присвоить имя, то в общем случае ему будет назначено имя целевого класса.

1. Статические (static) и динамические классы.

Язык UML позволяет представлять проектируемую систему с различных точек зрения. Различают три основных представления: структурная классификация, динамическое поведение и управление моделью.

Структурная классификация описывает системные сущности и их отношения между собой. В число классификаторов, имеющихся в моделях UML, входят классы, варианты использования, компоненты и узлы. Классификаторы являются базой, на которой строится динамическое поведение системы.

Динамическое поведение описывает поведение системы во времени. Поведение можно определить как ряд изменений в мгновенных снимках системы, полученных со статической точки зрения.

Представление управления моделью — это описание разбиения модели на иерархические блоки. Групповой организационный блок называется пакетом. Отдельные пакеты включают модели и подсистемы. Представление управления моделью организует все остальные представления моделей и позволяет осуществлять процесс разработки и управления конфигурацией.

В диаграмме классов основной акцент сделан на описании классов и их взаимоотношений: ассоциаций, обобщений и различных видов зависимостей, например реализаций и использования. Класс (class) представляет собой отдельную концепцию моделируемой системы. Классом может быть нечто, относящееся к материальному миру (например самолет), деловым отношениям (заказ), логике (расписание занятий) или поведению (задача). Класс описывает множество объектов со сходной структурой, поведением и отношениями. Классы — это основное понятие, вокруг которого строится объектно-ориентированная система.

Объекты, которые определяются классом, обладают состоянием и поведением. Состояние описывается атрибутами и ассоциациями. Атрибуты обычно используются для хранения значений, не обладающих индивидуальностью (например чисел и строк). Ассоциации нужны для связывания объектов, обладающих индивидуальностью. Отдельные элементы поведения описываются операциями класса. Реализацией операции является метод. История жизни объекта описывается конечным автоматом, который привязан к классу.

Графически класс изображается в виде прямоугольника. Атрибуты и операции класса перечисляются в горизонтальных отделениях этого прямоугольника.Отношения

между классами выражаются при помощи различных линий и дополнительных обозначений, которые ставятся либо над самими линиями, либо у их концов.

На диаграмме классов имеется три вида отношений: ассоциация (association), агрегация (aggregation) и обобщение (generalization).

Нотация класса в UML представляет собой прямоугольник, разделенный на три части, в которых задается имя класса, имена его атрибутов и операций. Атрибуты и операции могут быть помечены в соответствии с типом доступа и видимости.

При описании класса используются следующие соглашения UML:

- Статические (static) члены класса подчёркиваются.
- Описание операций имеет следующую форму:

<*спецификатор_доступа*><*имя_операции*> (<*список_параметров*>)
: <*тип_возвращаемого_значения*>

Спецификатор доступа (access specifier) может принимать одно из следующих значений (таблица.3.1).

Значения спецификаторов доступа

Таблица 3.1

Символ	**Тип доступа**
+	Public
-	Private
#	Protected

Список параметров (parameter list) - это последовательность параметров операции, разделенных запятыми. Форма каждого параметра:

<имя_параметра> :<тип_параметра>

Пример описания операции:

#MyOperation(param1 : integer, param2 : integer) : long

т.е. MyOperation() является защищенным (# - protected) членом класса, имеет два параметра целого типа (integer), тип возвращаемого значения – длинное целое (long).

2. Диаграммы объектов.

Для упрощения сложных диаграмм классов можно группировать классы в пакеты. Пакеты представляют собой наборы взаимосвязанных UML-элементов.

Диаграмма объектов показывает взаимосвязи экземпляров некоторых классов. Она используется для пояснения некоторых частей системы со сложными отношениями между объектами, особенно в случае использования рекурсивных отношений.

Каждый прямоугольник на диаграмме соответствует одному объекту заданного типа. Имена объектов подчеркнуты, они являются составными: в левой части, до двоеточия, находится собственно имя объекта, а в правой – имя класса, к которому принадлежит объект.

3. Диаграммы прецедентов.

Любые (в том числе и программные) системы проектируются с учетом того, что в процессе своей работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются**экторами**, причем каждый эктор ожидает, что система будет вести себя строго определенным, предсказуемым образом. Попробуем дать более строгое определение эктора. Для этого воспользуемся замечательным визуальным словарем по UML *ZicomMentor*:

Эктор (actor) - это множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Эктором может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности.

Графически эктор изображается либо " *человечком* ", подобным тем, которые мы рисовали в детстве, изображая членов своей семьи, либо*символом класса с соответствующим стереотипом*, как показано на рисунке. Обе формы представления имеют один и тот же смысл и могут использоваться в диаграммах. "Стереотипированная" форма чаще применяется для представления системных экторов или в случаях, когда эктор имеет свойства и их нужно отобразить.

Прецедент (use-case) - описание отдельного аспекта поведения системы с точки зрения пользователя (Буч).

Определение вполне понятное и исчерпывающее, но его можно еще немного уточнить, воспользовавшись тем же *ZicomMentor* 'ом:

Прецедент (usecase) - описание множества последовательных событий (включая варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецедент представляет поведение сущности, описывая взаимодействие между экторами и системой. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется.

Прецеденты обозначаются очень простым образом - в виде эллипса, внутри которого указано его название. *Прецеденты и экторы соединяются с помощью линий*. Часто на одном из концов линии изображают *стрелку*, причем *направлена она к тому, у кого запрашивают сервис*, другими словами, чьими услугами пользуются.

Прецеденты могут включать другие прецеденты, расширяться ими, наследоваться и т. д. *Все эти возможности мы здесь рассматривать не будем*. Как уже говорилось выше, цель этого обзора - просто научить читателя выделять диаграмму прецедентов, понимать ее назначение и смысл обозначений, которые на ней встречаются.

Из всего сказанного выше становится понятно, что *диаграммы прецедентов* относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное *средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями* продукта. Как мы уже могли убедиться, такие диаграммы очень просты для понимания и могут восприниматься и, что немаловажно, обсуждаться людьми, не являющимися специалистами в области разработки ПО.

Подводя итоги, можно выделить такие **цели создания диаграмм прецедентов**:

- определение границы и контекста моделируемой предметной области на ранних этапах проектирования;
- формирование общих требований к поведению проектируемой системы;
- разработка концептуальной модели системы для ее последующей детализации;
- подготовка документации для взаимодействия с заказчиками и пользователями системы.

4. Диаграмма состояний (конечных автоматов).

ля систем различной природы и назначения характерно взаимодействие между собой отдельных образующих их элементов. Для представления динамических особенностей взаимодействия элементов модели, в контексте реализации вариантов использования, предназначены диаграммы кооперации и последовательности. Однако для моделирования процессов функционирования большинства сложных систем, особенно систем реального времени, этих представлений недостаточно.

Диаграмма состояний (statechartdiagram) - диаграмма, которая представляет конечный автомат.

Семантика понятия *состояния* довольно сложна. Дело в том, что характеристика *состояний* системы явным образом не зависит от логической структуры, зафиксированной на диаграмме классов. При рассмотрении *состояний* системы приходится отвлекаться от особенностей ее объектной структуры и мыслить категориями,

описывающими динамический *контекст* поведения моделируемой системы. При построении *диаграмм состояний* необходимо использовать специальные понятия, о которых и пойдет речь в данной лекции.

Ранее отмечалось, что любая прикладная система характеризуется не только структурой составляющих ее элементов, но и некоторым поведением или функциональностью. Для общего представления функциональности моделируемой системы предназначены *диаграммы вариантов использования*, которые на концептуальном уровне описывают поведение системы в целом. Для того чтобы представить наиболее общее поведение на логическом уровне, следует ответить на вопрос: "В процессе какого поведения система реализует необходимую пользователям функциональность?".

Главное назначение *диаграммы состояний* - описать возможные последовательности *состояний* и *переходов*, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного *цикла*. *Диаграмма состояний* представляет динамическое поведение сущностей, на основе спецификации их реакции на восприятие некоторых конкретных *событий*. Системы, которые реагируют на внешние действия от других систем или от пользователей, иногда называют реактивными. Если такие действия инициируются в произвольные случайные моменты времени, то говорят об асинхронном поведении модели.

Диаграммы состояний чаще всего используются для описания поведения отдельных систем и подсистем. Они также могут быть применены для спецификации функциональности экземпляров отдельных классов, т.е. для моделирования всех возможных изменений *состояний* конкретных объектов. *Диаграмма состояний* по существу является графом специального вида, который служит для представления *конечного автомата*.

Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы для более детального представления *состояний* отдельных элементов модели. Для понимания семантики конкретной *диаграммы состояний* необходимо представлять особенности поведения моделируемой сущности, а также иметь общие сведения из теории *конечных автоматов*.

Конечный автомат (statemachine) - модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом действий, а также изменение его отдельных свойств.

В контексте языка *UML* понятие *конечного автомата* обладает дополнительной семантикой. Вершинами графа *конечного автомата* являются *состояния* и другие типы элементов модели, которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения *переходов* из *состояния* в *состояние*. *Конечный автомат* описывает поведение отдельного объекта в форме последовательности *состояний*, охватывающих все этапы его жизненного *цикла*, начиная от создания объекта и заканчивая его уничтожением. Каждая *диаграмма состояний* представляет собой *конечный автомат*.

Основными понятиями, характеризующими *конечный автомат*, являются *состояние* и *переход*. Ключевое различие между ними заключается в том, что длительность нахождения системы в отдельном *состоянии* существенно превышает время, которое затрачивается на *переход* из одного *состояния* в другое. Предполагается, что в пределе время *перехода* из одного *состояния* в другое равно нулю (если дополнительно ничего не сказано). Другими словами, *переход* объекта из *состояния* в *состояние* происходит мгновенно.

В общем случае *конечный автомат* представляет динамические аспекты моделируемой системы в виде ориентированного графа, вершины которого соответствуют *состояниям*, а дуги - *переходам*. При этом поведение моделируется как

последовательное перемещение по графу *состояний* от вершины к вершине по связывающим их дугам с учетом их ориентации. Для графа *состояний* системы можно ввести в рассмотрение специальные свойства.

Среди таких свойств - выделение из всей совокупности *состояний* двух специальных: начального и конечного. Ни в графе *состояний*, ни на *диаграмме состояний* время нахождения системы в том или ином *состоянии* явно не учитывается, однако предполагается, что последовательность изменения *состояний* упорядочена во времени. Другими словами, каждое последующее *состояние* может наступить позже предшествующего ему *состояния*.

Состояние (state) - **условие или ситуация в ходе жизненного цикла объекта, в течение которого он удовлетворяет логическому условию, выполняет определенную деятельность или ожидает события**.

Состояние может быть задано в виде набора конкретных значений атрибутов объекта некоторого класса, при этом изменение отдельных значений этих атрибутов будет отражать изменение *состояния* моделируемого объекта или системы в целом. Однако не каждый *атрибут* класса может характеризовать *состояние* его объектов. Как правило, имеют *значение* только те свойства элементов системы, которые отражают динамический или функциональный аспект ее поведения. В этом случае *состояние* будет характеризоваться некоторым инвариантным условием, включающим в себя только принципиальные для поведения объекта или системы атрибуты классов и их значения.

Такое условие может соответствовать ситуации, когда моделируемый *объект* находится в *состоянии* ожидания возникновения внешнего *события*. В то же время нахождение объекта в некотором *состоянии* может быть связано с выполнением определенных действий. В последнем случае соответствующая *деятельность* начинается в момент *перехода* моделируемого элемента в рассматриваемое *состояние*, а после и элемент может покинуть данное *состояние* в момент завершения этой деятельности.

Состояние на диаграмме изображается прямоугольником со скругленными вершинами. Этот *прямоугольник*, в свою *очередь*, может быть разделен на две секции горизонтальной линией. Если указана лишь одна секция, то в ней записывается только имя *состояния* (рис. 9.1, а). В противном случае в первой из них записывается имя *состояния*, а во второй - *список* некоторых внутренних действий или *переходов* в данном *состоянии* (рис. 9.1, б). При этом под действием в языке *UML* понимают некоторую *атомарную операцию*, выполнение которой приводит к изменению *состояния* или возврату некоторого значения (например, "*истина*" или "*ложь*").

Имя *состояния* представляет собой строку текста, которая раскрывает содержательный смысл или семантику данного *состояния*. Имя должно представлять собой законченное предложение и всегда записываться с заглавной буквы. Поскольку *состояние* системы является частью процесса ее функционирования, рекомендуется в качестве имени использовать глаголы в настоящем времени или соответствующие причастия. Как *исключение*, имя у *состояния* может отсутствовать, т.е. оно необязательно для некоторых *состояний*. В этом случае *состояние* является анонимным. Если на одной *диаграмме состояний* несколько анонимных *состояний*, то все они должны различаться между собой.

Действие (action) - спецификация выполнимого утверждения, которая образует абстракцию вычислительной процедуры.

Действие обычно приводит к изменению *состояния* системы, и может быть реализовано посредством передачи сообщения объекту, модификацией связи или значения атрибута. Для ряда *состояний* может потребоваться дополнительно указать действия, которые должны быть выполнены моделируемым элементом. Для этой цели

служит добавочная секция в обозначении *состояния*, содержащая перечень внутренних действий или *деятельность*, которые производятся в процессе нахождения моделируемого элемента в данном *состоянии*. Каждое действие записывается в виде отдельной строки и имеет следующий формат:

<метка действия '/ ' выражение действия>

Метка действия указывает на обстоятельства или условия, при которых будет выполняться *деятельность*, определенная выражением действия. При этом *выражение* действия может использовать любые атрибуты и связи, принадлежащие области имен или контексту моделируемого объекта. Если *список* выражений действия пустой, то *метка* действия с разделителем в виде наклонной черты '/' не указывается. Перечень меток действий в языке *UML* фиксирован, причем эти метки не могут быть использованы в качестве имен *событий*:

Входное действие (entryaction) - действие, которое выполняется в момент перехода в данное состояние.

Обозначается с помощью ключевого слова - метки действия entry, которое указывает на то, что следующее за ней *выражение* действия должно быть выполнено в момент входа в данное *состояние*.

Действие выхода (exitaction) - действие, производимое при выходе из данного состояния.

Обозначается с помощью ключевого слова - метки действия exit, которое указывает на то, что следующее за ней *выражение* действия должно быть выполнено в момент выхода из данного *состояния*.

Внутренняя деятельность (doactivity) - выполнение объектом операций или процедур, которые требуют определенного времени.

Обозначается с помощью ключевого слова - метки деятельности do, которое специфицирует так называемую "ду-*деятельность*", выполняемую в течение всего времени, пока *объект* находится в данном *состоянии*, или до тех пор, пока не будет прервано внешним *событием*. При нормальном завершении внутренней деятельности генерируется соответствующее *событие*.

Во всех остальных случаях *метка* действия идентифицирует *событие*, которое запускает соответствующее *выражение* действия. Эти *события* называются внутренними *переходами*. Семантически они эквивалентны *переходам* в само это *состояние*, за исключением той особенности, что *выход* из этого *состояния* или повторный вход в него не происходит. Это означает, что действия входа и выхода не производятся. При этом выполнение внутренних действий в *состоянии* не может быть прервано никакими внешними *событиями*, в отличие от внутренней деятельности, выполнение которой требует определенного времени.

В качестве примера *состояния* можно рассмотреть аутентификацию клиента для доступа к ресурсам моделируемой информационной системы (рис. 9.2). *Список* внутренних действий в данном *состоянии* может включать следующие действия. Первое действие - *входное*, которое выполняется при входе в это *состояние* и связано с получением строки символов, соответствующих паролю клиента. Далее выполняется *деятельность* по проверке введенного клиентом пароля. При успешном завершении этой проверки выполняется *действие на выходе*, которое отображает *меню* доступных для клиента опций.

Кроме обычных *состояний* на *диаграмме состояний* могут размещаться псевдосостояния.

Псевдосостояние (pseudo-state) - вершина в конечном автомате, которая имеет форму состояния, но не обладает поведением.

Примерами псевдосостояний, которые определены в языке *UML*, являются начальное и конечное *состояния*.

Начальное состояние (startstate) - разновидность псевдосостояния, обозначающее начало выполнения процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.

В этом *состоянии* находится *объект* по умолчанию в начальный момент времени. Оно служит для указания на *диаграмме состояний* графической области, от которой начинается процесс изменения *состояний*. Графически начальное *состояние* в языке *UML* обозначается в виде закрашенного кружка из которого может только выходить стрелка-переход.

На самом верхнем уровне представления объекта *переход* из начального *состояния* может быть помечен *событием* создания (инициализации) данного объекта. В противном случае этот *переход* никак не помечается. Если этот *переход* не помечен, то он является первым *переходом* на *диаграмме состояний* в следующее за ним *состояние*. Каждая *диаграмма* или под-*диаграмма состояний* должна иметь единственное начальное *состояние*.

Конечное состояние (finalstate) - разновидность псевдосостояния, обозначающее прекращение процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.

В этом *состоянии* должен находиться моделируемый *объект* или система по умолчанию после завершения работы *конечного автомата*. Оно служит для указания на *диаграмме состояний* графической области, в которой завершается процесс изменения *состояний* или *жизненный цикл* данного объекта. Графически конечное *состояние* в языке *UML* обозначается в виде закрашенного кружка, помещенного в *окружность*, в которую может только входить стрелка-переход. Каждая *диаграмма состояний* или *подсостояний* может иметь несколько конечных *состояний*, при этом все они считаются эквивалентными на одном уровне вложенности *состояний*.

5. Диаграммы последовательностей.

Диаграммы последовательностей - это отличное средство документирования поведения системы, детализации логики сценариев использования; но есть еще один способ - использовать диаграммы взаимодействия. Диаграмма взаимодействия *показывает поток сообщений между объектами системы и основные ассоциации между ними* и по сути, как уже было сказано выше, является альтернативой диаграммы последовательностей. Внимательный читатель, возможно, скажет, что *диаграмма объектов* делает то же самое, - и будет не прав. *Диаграмма объектов показывает статику*, некий снимок системы, связи между объектами в данный момент времени, диаграмма же взаимодействия, как и диаграмма последовательностей, показывает взаимодействие (извините за невольный каламбур) объектов во времени, т. е. в динамике.

Следует отметить, что использование диаграммы последовательностей или диаграммы взаимодействия - личный выбор каждого проектировщика и зависит от индивидуального стиля проектирования. Мы, например, чаще отдаем предпочтение диаграмме последовательностей. На обозначениях, применяемых на диаграмме взаимодействия, думаем, не стоит останавливаться подробно. Здесь все стандартно: объекты обозначаются прямоугольниками с подчеркнутыми именами (чтобы отличить их от классов, помните?), ассоциации между объектами указываются в виде соединяющих их линий, над ними может быть изображена стрелка с указанием названия сообщения и его порядкового номера.

Необходимость номера сообщения объясняется очень просто - в отличие от диаграммы последовательностей, *время на диаграмме взаимодействия не показывается в виде отдельного измерения*. Поэтому последовательность передачи сообщений можно

указать только с помощью их нумерации. В этом и состоит вероятная причина пренебрежения этим видом диаграмм многими проектировщиками.

6. Диаграммы коммуникации (взаимодействия).

Диаграмма коммуникации (англ. *communication diagram*, в UML 1.x — диаграмма кооперации, collaboration diagram) —диаграмма, на которой изображаются взаимодействия между частями композитной структуры или ролями кооперации. В отличие от диаграммы последовательности, на диаграмме коммуникации явно указываются отношения между объектами, а время как отдельное измерение не используется (применяются порядковые номера вызовов).

В UML есть четыре типа диаграмм взаимодействия:

- Диаграмма последовательности
- Диаграмма коммуникации
- Диаграмма обзора взаимодействия
- Диаграмма синхронизации

Диаграмма коммуникации моделирует взаимодействия между объектами или частями в терминах упорядоченных сообщений. Коммуникационные диаграммы представляют комбинацию информации, взятой из диаграмм классов, последовательности и вариантов использования, описывая сразу и статическую структуру и динамическое поведение системы.

Коммуникационные диаграммы имеют свободный формат упорядочивания объектов и связей как в диаграмме объектов. Чтобы поддерживать порядок сообщений при таком свободном формате, их хронологически нумеруют. Чтение диаграммы коммуникации начинается с сообщения 1.0 и продолжается по направлению пересылки сообщений от объекта к объекту.

Диаграмма коммуникации показывает во многом ту же информацию, что и диаграмма последовательности, но из-за другого способа представления информации какие-то вещи на одной диаграмме видеть проще, чем на другой. Диаграмма коммуникаций нагляднее показывает, с какими элементами взаимодействует каждый элемент, а диаграмма последовательности яснее показывает в каком порядке происходят взаимодействия.

7. Зачем так много различных диаграмм?

В процессе проектирования система рассматривается **с разных точек зрения** с помощью моделей, различные представления которых предстают в форме диаграмм. **Модель** - это некий (материальный или нет) *объект*, отображающий лишь наиболее значимые для данной задачи характеристики системы. Модели бывают разные - материальные и нематериальные, искусственные и естественные, декоративные и математические...

Несмотря на то что в предыдущем абзаце мы весьма вольготно обошлись с понятием модели, следует понимать, что в контексте приведенных выше определений **ни одна отдельная диаграмма не является моделью**. Диаграммы - лишь средство визуализации модели, и эти два понятия следует различать. Лишь **набор диаграмм составляет модель системы** и наиболее полно ее описывает, но не одна *диаграмма*, вырванная из контекста.

UML 1.5 определял **двенадцать типов диаграмм**, разделенных на три группы:

- четыре типа диаграмм представляют статическую структуру приложения;
- пять представляют поведенческие аспекты системы;
- три представляют физические аспекты функционирования системы (диаграммы реализации).

Текущая версия *UML* 2.1 внесла не слишком много изменений. Диаграммы слегка изменились внешне (появились фреймы и другие визуальные улучшения), немного усовершенствовалась *нотация*, некоторые диаграммы получили новые наименования.

Виды диаграмм:

- *диаграмма прецедентов*;
- диаграмма классов;
- *диаграмма объектов*;
- диаграмма последовательностей;
- диаграмма взаимодействия;
- диаграмма состояний;
- *диаграмма активности*;
- *диаграмма развертывания*.

1. Диаграммы видов деятельности.

Диаграмма видов деятельности (англ. *activitydiagram*) — UML-диаграмма, на которой показано разложение некоторой деятельности на её составные части. Под деятельностью (англ. *activity*) понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий англ. *action*, соединённых между собой потоками, которые идут от выходов одного узла ко входам другого.

Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений.

Диаграммы деятельности состоят из ограниченного количества фигур, соединённых стрелками. Основные фигуры:

1. Прямоугольники с закруглениями — действия
2. Ромбы — решения
3. Широкие полосы — начало (разветвление) и окончание (схождение) ветвления действий
4. Чёрный круг — начало процесса (начальное состояние)
5. Чёрный круг с обводкой — окончание процесса (конечное состояние)

Стрелки идут от начала к концу процесса и показывают последовательность переходов.

Диаграмма деятельности – еще один способ описания поведения, который визуально напоминает старую добрую блок-схему алгоритма. Однако за счет модернизированных обозначений, согласованных с объектно-ориентированным подходом, а главное, за счет новой семантической составляющей (свободная интерпретация сетей Петри), диаграмма деятельности UML является мощным средством для описания поведения системы.

На диаграмме деятельности применяют один основной тип сущностей – действие, и один тип отношений – переходы (передачи управления и данных). Также используются такие конструкции как развилки, слияния, соединения, ветвления, которые похожи на сущности, но таковыми на самом деле не являются, а представляют собой графический способ изображения некоторых частных случаев многоместных отношений.

2. Диаграммы компонентов.

Полный проект программной системы представляет собой совокупность моделей логического и физического уровней, которые должны быть согласованы между собой. В языке UML для физического представления моделей систем используются диаграммы реализации (implementationdiagrams), которые включают в себя ***диаграмму компонентов*** и ***диаграмму развертывания***.

Диаграмма компонентов, **Componentdiagram** — статическая структурная диаграмма, показывает разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами. В качестве физических компонентов могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Она позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный и исполняемый код. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

Диаграмма компонентов разрабатывается для следующих целей:

- визуализации общей структуры исходного кода программной системы;
- спецификации исполняемого варианта программной системы;
- обеспечения многократного использования отдельных фрагментов программного кода;
- представления концептуальной и физической схем баз данных.

В разработке диаграмм компонентов участвуют как системные аналитики и архитекторы, так и программисты. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут существовать только на этапе компиляции программного кода, другие на этапе его исполнения. Диаграмма компонентов отражает общие зависимости между компонентами, рассматривая последние в качестве классификаторов.

Компоненты

Для представления физических сущностей в языке UML применяется специальный термин - ***компонент*** (component). Компонент реализует некоторый набор интерфейсов и служит для общего обозначения элементов физического представления модели. Для графического представления компонента используется специальный символ - прямоугольник со вставленными слева двумя более мелкими прямоугольниками. Внутри большого прямоугольника записывается имя компонента и, при необходимости, некоторая дополнительная информация. Изображение этого символа может незначительно варьироваться в зависимости от характера ассоциируемой с компонентом информации.

Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и некоторых знаков препинания.

Отдельный компонент может быть представлен на уровне типа или на уровне экземпляра. Графическое изображение в обоих случаях одинаковое, но правила записи имени компонента отличаются. Если компонент представляется на уровне типа, то в качестве его имени записывается только имя типа с заглавной буквы. Если же компонент представляется на уровне экземпляра, то в качестве его имени записывается <имя компонента>':'<имя типаX>. При этом вся строка имени подчеркивается.

В качестве простых имен принято использовать имена исполняемых файлов (с указанием расширения exe после точки-разделителя), динамических библиотек (расширение dll), Web-страниц (расширение html), текстовых файлов (расширения txt или doc) или файлов справки (hip), файлов баз данных (DB) или файлов с исходными текстами программ (расширения h, cpp для языка C++, расширение java для языка Java), скрипты (pi, asp) и другие.

Поскольку конкретная реализация логического представления модели системы зависит от используемого программного инструментария, то и имена компонентов определяются особенностями синтаксиса соответствующего языка программирования.

В отдельных случаях к простому имени компонента может быть добавлена информация об имени объемлющего пакета и о конкретной версии реализации данного компонента. В этом случае номер версии записывается как помеченное значение в фигурных скобках. В других случаях символ компонента может быть разделен на секции, чтобы явно указать имена реализованных в нем интерфейсов.

Поскольку компонент как элемент физической реализации модели представляет отдельный модуль кода, иногда его комментируют с указанием дополнительных графических символов, иллюстрирующих конкретные особенности его реализации. Эти дополнительные обозначения для примечаний не специфицированы в языке UML, однако их применение упрощает понимание диаграммы компонентов, повышая наглядность физического представления.

В языке UML выделяют ***три вида компонентов***:

- развертывания, которые обеспечивают непосредственное выполнение системой своих функций. Такими компонентами могут быть динамически подключаемые библиотеки с расширением dll, Web-страницы на языке разметки гипертекста с расширением html и файлы справки с расширением hlp;
- рабочие продукты. Как правило, это файлы с исходными текстами программ, например, с расширениями h или cpp для языка C++;
- исполнения, представляющие собой исполняемые модули - файлы с расширением exe.

Эти элементы иногда называют артефактами, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов.

Другим способом спецификации различных видов компонентов является явное указание его стереотипа компонента перед именем. В языке UML для компонентов определены следующие стереотипы:

- библиотека (library) - определяет первую разновидность компонента, который представляется в форме динамической или статической библиотеки;
- таблица (table) - также определяет первую разновидность компонента, который представляется в форме таблицы базы данных;
- файл (file) - определяет вторую разновидность компонента, который представляется в виде файлов с исходными текстами программ;
- документ (document) - определяет вторую разновидность компонента, . который представляется в форме документа;
- исполнимый (executable) — определяет третий вид компонента, который может исполняться в узле.

Интерфейсы

Следующим элементом диаграммы компонентов являются ***интерфейсы***. В общем случае, интерфейс графически изображается окружностью, которая соединяется с компонентом отрезком линии без стрелок. Имя интерфейса должно начинаться с заглавной буквы "**I**" и записываться рядом с окружностью. Семантически линия означает реализацию интерфейса, а наличие интерфейсов у компонента означает, что данный компонент реализует соответствующий набор интерфейсов.

Другим способом представления интерфейса на диаграмме компонентов является его изображение в виде прямоугольника класса со стереотипом «интерфейс» и возможными секциями атрибутов и операций. Как правило, этот вариант обозначения используется для представления внутренней структуры интерфейса, которая может быть важна для реализации.

При разработке программных систем интерфейсы обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программы, не изменяя другие ее части. Таким образом, назначение

интерфейсов существенно шире, чем спецификация взаимодействия с пользователями системы (актерами).

Зависимости

В общем случае отношение ***зависимости*** также было рассмотрено ранее. Напомним, что зависимость не является ассоциацией, а служит для представления только факта наличия такой связи, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. Отношение зависимости на диаграмме компонентов изображается пунктирной линией со стрелкой, направленной от клиента (зависимого элемента) к источнику (независимому элементу).

Зависимости могут отражать связи модулей программы на этапе компиляции и генерации объектного кода. В другом случае зависимость может отражать наличие в независимом компоненте описаний классов, которые используются в зависимом компоненте для создания соответствующих объектов. Применительно к диаграмме компонентов зависимости могут связывать компоненты и импортируемые этим компонентом интерфейсы, а также различные виды компонентов между собой.

В первом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу. Наличие стрелки означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения. Причем на этой же диаграмме может присутствовать и другой компонент, который реализует этот интерфейс.

Другим случаем отношения зависимости на диаграмме компонентов является отношение между различными видами компонентов. Наличие подобной зависимости означает, что внесение изменений в исходные тексты программ или динамические библиотеки приводит к изменениям самого компонента. При этом характер изменений может быть отмечен дополнительно.

На диаграмме компонентов могут быть также представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет значение для обеспечения согласования логического и физического представлений модели системы. Если требуется подчеркнуть, что некоторый компонент реализует отдельные классы, то для обозначения компонента используется расширенный символ прямоугольника. При этом прямоугольник компонента делится на две секции горизонтальной линией. Верхняя секция служит для записи имени компонента, а нижняя секция — для указания дополнительной информации.

Внутри символа компонента могут изображаться другие элементы графической нотации, такие как классы (компонент уровня типа) или объекты (компонент уровня экземпляра). В этом случае символ компонента изображается таким образом, чтобы вместить эти дополнительные символы.

Объекты, которые находятся в отдельном компоненте-экземпляре, изображаются вложенными в символ данного компонента. Подобная вложенность означает, что выполнение компонента влечет выполнение соответствующих объектов.

3. Диаграммы развертывания.

Диагра́ммаразвёртывания,**Deploymentdiagram**вUMLмоделирует*физическое*разве ртывание артефактов наузлах.[1] Например, чтобы описать веб-сайт диаграмма развертывания должна показывать, какие аппаратные компоненты («узлы») существуют (например, веб-сервер, сервер базы данных, сервер приложения), какие программные компоненты («артефакты») работают на каждом узле (например, веб-приложение, база данных), и как различные части этого комплекса соединяются друг с другом (например, JDBC, REST, RMI).

Узлы представляются как прямоугольные параллелепипеды с артефактами, расположенными в них, изображенными в виде прямоугольников. Узлы могут иметь подузлы, которые представляются как вложенные прямоугольные параллелепипеды. Один

узел диаграммы развертывания может концептуально представлять множество физических узлов, таких как кластер серверов баз данных.

Существует два типа узлов:

1. Узел устройства
2. Узел среды выполнения

Узлы устройств — это физические вычислительные ресурсы со своей памятью и сервисами для выполнения программного обеспечения, такие как обычные ПК, мобильные телефоны. Узел среды выполнения — это программный вычислительный ресурс, который работает внутри внешнего узла и который предоставляет собой сервис, выполняющий другие исполняемые программные элементы.

4. Диаграммы пакетов.

Диаграммы пакетов унифицированного языка моделирования(UML) отображают зависимости между пакетами, составляющими модель. В дополнение к стандартным отношениям зависимостей в UML есть два специальных вида зависимостей, определенных между пакетами:

- Импортирование пакета
- Слияние пакета

Импортирование пакета — это отношение между импортирующим пространством имен и пакетом, указывающим на то, что импортирующее пространство имен добавляет имена членов пакета в их собственное пространство имен. По умолчанию, непомеченная зависимость между двумя пакетами интерпретируется как отношение «импорт пакета» ..

Слияние пакета — направленное отношение между двумя пакетами, которое указывает, что содержимое двух пакетов должно быть объединено. Это очень похоже на «Обобщение» в том смысле, что исходный элемент как бы добавляет характеристики целевого элемента к своим собственным характеристикам, в результате чего получается элемент, который сочетает в себе характеристики обоих элементов.

Диаграммы пакетов могут использовать пакеты, содержащие прецеденты для иллюстрации функциональности программного обеспечения системы. Диаграммы могут использовать пакеты, которые представляют различные слои программного комплекса для иллюстрации его слоистой архитектуры. Зависимости между этими пакетами могут быть снабжены метками / стереотипами, чтобы указать механизм связи между слоями.

5. Временная диаграмма.

Временная диаграмма - это изображение, представляющее определенный период времени и события, происходящие в этот период.*Временные диаграммы* особенно удобны при отображении общего хода проекта - его состояния, истории события и того, что должно быть сделано. Обычно *временные диаграммы* включают важные события и маркеры интервалов.

Диаграммы расписания проекта могут содержать на странице одну или несколько временных диаграмм, которые могут быть синхронизированы друг с другом.

Обычно одна из временных диаграмм является главной, а остальные представляют собой расширенные *временные диаграммы*. **Главную временную диаграмму** можно представить как полное высокоуровневое *представление* событий, а **расширенную временную диаграмму** - как более подробное *представление* временного периода.

Расширенную временную диаграмму используют для представления отрезка **главной временной диаграммы** для отображения большего количества информации об этом отрезке времени. Важные события или интервалы добавляются на расширенную временную диаграмму точно так же, как и на главную. Элементы, добавляемые на расширенную временную диаграмму, не отображаются на главной, но любая фигура, добавляемая на главную временную диаграмму, отображается на расширенной и синхронизируется с ней.

Временная *диаграмма* содержит:

- Основные события - значительные события и даты, такие как дата завершения этапа проекта.
- Маркеры интервалов - указывают длительность временных периодов.

2. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ПРОВЕДЕНИЮ ПРАКТИЧЕСКИХЗАНЯТИЙ

2.1 Практическое занятие №1, 2, 3, 4 (8 часов).

Тема: «Общие принципы проектирования информационных систем**»**

2.1.1 Задание для работы:

1. Изучить объект проектирования.
2. Изучить процесс проектирования.
3. Понять формальную логику.
4. Изучить процесс образования понятия.
5. Изучить диалектическую логику.
6. Изучить содержательно-генетическую логику.
7. Изучить логическую схему проектирования.
8. Изучить логические отношения.
9. Понять задачи проектирования.
10. Изучить определение логической схемы проектирования (методологии проектирования).
11. Ознакомиться с решением задач проектирования.

2.1.2 Краткое описание проводимого занятия:

Рассматриваются понятия проектирование, объекта, предмета проектирования. Стадии и этапы проектирования. Понятие формальной логики и ее отличие от неформальной. Процесс образования понятия. Понятие и предмет диалектической логики. Положения концепции содержательно-генетической логики. Логическая схема проектирования. Алгоритм проектирования. Результат проектирования. Проектное решение. Простые и сложные суждения. Задачи проектирования. Схема логического проектирования. Этапы методологии логического проектирования. Решения задач проектирования.

2.1.3 Результаты и выводы:

Различные виды проектирования ориентированы на создание и преобразование разных объектов и предметов. Объект и предмет проектирования соотносятся между собой как общее и частное.

Комплекс проектных работ включает в себя теоретические и экспериментальные исследования, расчеты, конструирование.

Проектирование разделяется на этапы.

Формальная логика, в отличие от неформальной, организована как формальная система, обладающая высоким уровнем абстракции и чётко определёнными методами, правилами и законами.

Приемами образования понятия являются: абстрагирование, анализ, синтез, сравнение и обобщение. Все логические приемы образования понятий имеют важнейшее значение. Они связаны между собой, их невозможно представить один без другого. Часто применяются вместе или предшествуют один другому.

Диалектическая логика - это учение о познании, о философском постижении объективной истины. Она описывает процесс познания не реальной сферы действительности, а абстрактного объекта. Содержание диалектической логики показывает диалектический метод философского познания в чистом, наиболее общем, абстрактном виде. Формальная логика исследует отношения между мыслями, выраженные в стабильных, неизменных структурах.

Процесс мышления протекает согласно логическим законам независимо от того, знаем мы об их существовании или нет. Вследствие своей объективности логические законы, так же как и физические, нельзя нарушить, отменить или переиначить.

Применение многозначных логических и запоминающих элементов при создании промышленных систем управления обусловливает необходимость разработки соответствующих методов синтеза подобных систем. Для проектирования устройств с многозначным структурным алфавитом в общем случае оказывается неприменимым тот аппарат, который используется для синтеза устройств с двузначным структурным алфавитом. В связи с этим возникает необходимость как разработки специального аппарата, который был бы пригоден для проектирования логических схем на многозначных структурах, так и построения общих методов синтеза, не зависящих от значности применяемой логики.

Основу отношений между суждениями составляет их сходство по смыслу и логическим значениям (истинности и ложности). В силу этого отношения устанавливаются не между любыми, а лишь между сравнимыми, т.е. имеющими общий смысл, суждениями. Сложные суждения также могут быть сравнимыми и несравнимыми.

Несравнимые — это суждения, которые не имеют общих пропозициональных переменных.

Сравнимые — это суждения, которые имеют одинаковые пропозиционные переменные (составляющие) и различаются логическими связками, включая отрицание.

Сложные сравнимые суждения могут быть совместимыми и несовместимыми.

На стадии проектирования проектировщик должен проделать определенную работу, удовлетворяющую задачам проектирования.

Методология логического проектирования подразделяется на этапы.

Одним из путей предсказания поведения проектируемых систем является путь создания математических моделей и последующего проведения исследования систем на этих моделях. Построение или проектирование систем, удовлетворяющих заранее заданным свойствам, можно осуществить, когда имеются управляющие переменные, при помощи которых можно влиять на поведение проектируемой системы.

2.2 Практическое занятие №5, 6, 7, 8 (8 часов).

Тема: «Виды моделей компонентов информационных систем**»**

2.2.1 Задание для работы:

1. Изучить понятие системного подхода и системного анализа.
2. Изучить документ. Электронный документ. Информационная система.
3. Изучить информационную технологию.
4. Изучить модели жизненного цикла информационных систем.
5. Изучить каскадную модель.

2.2.2 Краткое описание проводимого занятия:

Основной общий принцип системного подхода. Трактовки понятия системного анализа. Определение документа, электронного документа, понятие и назначение информационной системы. Понятие информационной технологии. Понятие жизненного цикла информационных систем. Виды моделей жизненного цикла. Понятие каскадной модели. Преимущества и недостатки каскадной модели.

2.2.3 Результаты и выводы:

Системный подход включает в себя выявление структуры системы, типизацию связей, определение атрибутов, анализ влияния внешней среды.

Системный анализ характеризуется главным образом упорядоченным, логически обоснованным подходом к исследованию проблем и использованию существующих методов их решения, которые могут быть разработаны в рамках других наук.

Целью системного анализа является полная и всесторонняя проверка различных вариантов действий с точки зрения количественного и качественного сопоставления затраченных ресурсов с получаемым эффектом.

Документ является основным способом представления информации. ***Электронный документ*** — это бумажный документ, введённый в *компьютер* для обработки.

Информационная система предназначена для своевременного обеспечения надлежащих людей надлежащей информацией, то есть для удовлетворения конкретных информационных потребностей в рамках определенной предметной области, при этом результатом функционирования информационных систем является информационная продукция — документы, информационные массивы, базы данных и информационные услуги.

Информационная технология – процесс, использующий совокупность средств методов сбора, обработки и передачи первичной информации для получения информации нового качества о состоянии объекта, т.е. информационного продукта.

Информационные технологии состоят из этапов, каждый из них включает операции, а последние состоят из элементарных действий, таких как нажатие какой-нибудь клавиши, выбор позиции в меню и т.д.

Жизненный цикл ИС можно представить как ряд событий, происходящих с системой в процессе ее создания и использования.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления. Модель жизненного цикла - структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

Каскадная модель жизненного цикла предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

2.3 Практическое занятие №9, 10, 11, 12 (8 часов).

Тема: «Виды систем проектирования АСОИ**»**

2.3.1 Задание для работы:

1. Изучить унифицированный процесс – управляемый вариантами использования.
2. Изучить унифицированный процесс - ориентирован на архитектуру.
3. Изучить унифицированный процесс - итеративный и инкрементный.
4. Изучить жизненный цикл в унифицированном процессе.
5. Изучить продукт унифицированного процесса.
6. Изучить унифицированный процесс – методология разработки.

2.3.2 Краткое описание проводимого занятия:

Унифицированный процесс разработки программного обеспечения с точки зрения управляемости вариантами использования. Унифицированный процесс с точки зрения ориентации на архитектуру. Итеративный и инкрементный унифицированный процесс. Фазы жизненного цикла в унифицированном процессе. Что включает в себя продукт унифицированного процесса. Модели унифицированного процесса. Методология разработки унифицированного процесса.

2.3.3 Результаты и выводы:

Унифицированный процесс есть процесс разработки программного обеспечения. Процесс разработки программного обеспечения - это сумма различных

видов деятельности, необходимых для преобразования требований пользователей в программную систему. Однако Унифицированный процесс - это больше, чем единичный процесс, это обобщенный каркас процесса, который может быть специализирован для широкого круга программных систем, различных областей применения, уровней компетенции и размеров проекта.

Унифицированный процесс компонентно-ориентирован. Это означает, что создаваемая программная система строится на основе программных компонентов, связанных хорошо определенными интерфейсами.

Для разработки чертежей программной системы Унифицированный процесс использует Унифицированный язык моделирования.

Архитектура - это представление всего проекта с выделением ключевых составляющих и затушевывание деталей. Архитектура вырастает из требований к результату, в том виде, как их понимает пользователь и другие заинтересованные лица.

Каждый продукт имеет функции и форму, причем одно без другого не существует. В нашем случае функции, как мы ранее отмечали, соответствуют вариантам использования, а форма – архитектуре. Согласно знаниям, заложенным в методологии (унифицированного процесса),сначала должны быть разработаны варианты использования, то есть функции, а потом для того чтобы обеспечить выполнение этих функций разрабатывается архитектура системы. С другой стороны архитектура должна обеспечить реализацию необходимых сейчас и в будущем функций, то есть вариантов использования. Реально архитектура и варианты использования разрабатываются параллельно.

Таким образом, архитектор придает системе форму и архитектор, проектируя форму, должен заложить такие решения, которые бы позволили системе развиваться не только в момент начальной разработки, но и в будущих поколениях системы.

На каждой итерации разработчики определяют и описывают уместные варианты использования, создают проект, использующий выбранную архитектуру в качестве направляющей, реализуют проект в компоненты и проверяют соответствие компонентов вариантам использования. Если итерация достигла своей цели, процесс разработки переходит на следующую итерацию. Если итерация не выполнила своей задачи, разработчики должны пересмотреть свои решения и попробовать другой подход.

Унифицированный процесс циклически повторяется. Эта последовательность повторений Унифицированного процесса представляет собой жизненный цикл системы. Каждый цикл завершается поставкой выпуска продукта заказчикам.

Каждый цикл состоит из четырех фаз - анализа и планирования требований, проектирования, построения и внедрения. Каждая фаза, как будет рассмотрено ниже, далее подразделяется на итерации.

Каждый цикл осуществляется в течение некоторого времени. Это время, в свою очередь, делится на четыре фазы: фазу анализа и планирования требований, фазу проектирования, фазу построения и фазу внедрения. Внутри каждой фазы руководители или разработчики могут потерпеть неудачу в работе - но только на данной итерации и в связанном с ней приращении. Каждая фаза заканчивается вехой. Мы определяем каждую веху по наличию определенного набора артефактов, например, некая модель документа должна быть приведена в предписанное состояние.

Результатом каждого цикла является новый выпуск системы, а каждый выпуск - это продукт, готовый к поставке. Он включает в себя тело - исходный код, воплощенный в компоненты, которые могут быть откомпилированы и выполнены, плюс руководство и дополнительные компоненты поставки. Однако готовый продукт должен также быть приспособлен для нужд не только пользователей, а всех заинтересованных лиц. Программному продукту следовало бы представлять собой нечто большее, чем исполняемый машинный код.

Окончательный продукт включает в себя требования, варианты использования, нефункциональные требования и варианты тестирования. Он включает архитектуру и визуальные модели - артефакты, смоделированные на Унифицированном языке моделирования. Эти средства позволяют заинтересованным лицам использовать систему и модифицировать ее от поколения к поколению.

Для эффективного применения этих идей необходимо, чтобы они образовывали единый многоплановый процесс, поддерживающий циклы, фазы, рабочие процессы, снижение рисков, контроль качества, управление проектом и конфигурацией. Унифицированный процесс создает каркас, объединяющий все аспекты. Такой комплекс знаний, называют методологией разработки.

2.4 Практическое занятие №13, 14, 15, 16, 17 (10 часов).

Тема: «Типы диаграмм в языке UML**»**

2.4.1 Задание для работы:

1. Изучить классификацию диаграмм, принятые обозначения.
2. Изучить изображение ассоциаций на диаграммах классов.
3. Иерархии классов.
4. CRC-карточки.
5. Диаграмма классов.
6. Статические (static) и динамические классы.
7. Диаграммы объектов.
8. Диаграммы прецедентов.
9. Диаграмма состояний (конечных автоматов).
10. Диаграммы последовательностей.
11. Диаграммы коммуникации (взаимодействия).
12. Зачем так много различных диаграмм?
13. Диаграммы видов деятельности.
14. Диаграммы компонентов.
15. Диаграммы развертывания.
16. Диаграммы пакетов.
17. Временная диаграмма.

2.4.2 Краткое описание проводимого занятия:

Диаграмма UML. Структурные и поведенческие диаграммы. Ассоциации на диаграмме классов. Порядок наследования отношений в иерархии классов. Назначение CRC-карточек. Цель построения диаграммы классов. Описание статических и динамических классов. Определение диаграммы объектов, назначение и представление диаграммы. Понятие и цели создания диаграммы прецедентов. Цель создания диаграммы состояний. Понятие диаграммы последовательности. Назначение диаграммы коммуникации. Отличительные особенности. Использование диаграммы видов деятельности. Назначение диаграммы компонентов. Назначение диаграммы развертывания, пакетов и временной диаграммы. Узлы диаграммы развертывания. Расширенная временная диаграмма.

2.4.3 Результаты и выводы:

Рассматривая диаграмму UML, необходимо помнить, что основной принцип UML заключается в том, что любая информация на конкретной диаграмме может быть подавлена. Это подавление может носить глобальный характер – скрыть все атрибуты – или локальный – не показывать какие-нибудь конкретные классы. Поэтому по диаграмме вы никогда не можете судить о чем-нибудь по его отсутствию. Даже если метамодель UML имеет поведение по умолчанию, например для атрибутов, когда вы не видите эту

информацию на диаграмме, это может быть обусловлено либо поведением по умолчанию, либо тем, что она просто подавлена.

Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.

Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.

Самым распространенным видом отношения зависимости является соединение между классами, когда один класс использует другой в качестве параметра операции.

Моделирование отношений наследования осуществляется в таком порядке:

1. Найти атрибуты, операции и обязанности, общие для двух или более классов из данной совокупности
2. Вынести эти элементы в некоторый общий класс (если надо, создайте новый, но следите, чтобы уровней не оказалось слишком много).
3. Отметить в модели, что более специализированные классы наследуют более общим, включив отношение обобщения, направленное от каждого потомка к его родителю.

CRC-карточки - эффективный способ анализа сценариев. Карточки можно раскладывать так, чтобы представить формы сотрудничества объектов. С точки зрения динамики сценария, их расположение может показать поток сообщений между объектами, с точки зрения статики они представляют иерархии классов.

Диаграмма классов описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами.

Структурная классификация описывает системные сущности и их отношения между собой. В число классификаторов, имеющихся в моделях UML, входят классы, варианты использования, компоненты и узлы. Классификаторы являются базой, на которой строится динамическое поведение системы.

Динамическое поведение описывает поведение системы во времени. Поведение можно определить как ряд изменений в мгновенных снимках системы, полученных со статической точки зрения.

Диаграмма объектов показывает взаимосвязи экземпляров некоторых классов. Она используется для пояснения некоторых частей системы со сложными отношениями между объектами, особенно в случае использования рекурсивных отношений.

Диаграммы прецедентов относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями продукта. Такие диаграммы очень просты для понимания и могут восприниматься и, что немаловажно, обсуждаться людьми, не являющимися специалистами в области разработки ПО.

Главное назначение диаграммы состояний - описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла. Диаграмма состояний представляет динамическое поведение сущностей, на основе спецификации их реакции на восприятие некоторых конкретных событий. Системы, которые реагируют на внешние действия от других систем или от пользователей, иногда называют реактивными.

Если такие действия инициируются в произвольные случайные моменты времени, то говорят об асинхронном поведении модели.

Диаграммы последовательностей - это отличное средство документирования поведения системы, детализации логики сценариев использования; но есть еще один способ - использовать диаграммы взаимодействия.

Диаграмма коммуникации показывает во многом ту же информацию, что и диаграмма последовательности, но из-за другого способа представления информации какие-то вещи на одной диаграмме видеть проще, чем на другой. Диаграмма коммуникаций нагляднее показывает, с какими элементами взаимодействует каждый элемент, а диаграмма последовательности яснее показывает в каком порядке происходят взаимодействия.

Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений.

Диаграммы деятельности состоят из ограниченного количества фигур, соединённых стрелками.

Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут существовать только на этапе компиляции программного кода, другие на этапе его исполнения. Диаграмма компонентов отражает общие зависимости между компонентами, рассматривая последние в качестве классификаторов.

Диаграмма развёртывания моделирует физическое развертывание артефактов на узлах

Узлы устройств — это физические вычислительные ресурсы со своей памятью и сервисами для выполнения программного обеспечения, такие как обычные ПК, мобильные телефоны. Узел среды выполнения — это программный вычислительный ресурс, который работает внутри внешнего узла и который предоставляет собой сервис, выполняющий другие исполняемые программные элементы.

Диаграммы пакетов могут использовать пакеты, содержащие прецеденты для иллюстрации функциональности программного обеспечения системы. Диаграммы могут использовать пакеты, которые представляют различные слои программного комплекса для иллюстрации его слоистой архитектуры. Зависимости между этими пакетами могут быть снабжены метками / стереотипами, чтобы указать механизм связи между слоями.

Временные диаграммы особенно удобны при отображении общего хода проекта - его состояния, истории события и того, что должно быть сделано. Обычно временные диаграммы включают важные события и маркеры интервалов.