{ПРОГРАММИРОВАНИЕ}

ШАГ компьютерная АКАДЕМИЯ

UNIFIED MODELING LANGUAGE

Урок №1

Введение в объектно-ориентированный анализ и проектирование

Содержание

1. Сложности при разработке программного обеспечения

На сегодняшний день очень возросла сложность создаваемых систем.

Проявлениями этой сложности являются не только увеличение размеров и функциональности. Не меньше, если не больше, проблем порождается частой сменой потребностей пользователей и ростом требований к качеству систем.

Традиционные способы разрешения проблемы сложности, такие как увеличение коллективов разработчиков, их специализация, распределение работ в чистом виде, ведут к еще большим трудностям согласования результатов и сборки готовых систем.

Серьезным шагом к победе над сложностью явилось появление объектно-ориентированного программирования (ООП). Но только шагом. Появление ООП не устранило проблем недостаточного взаимопонимания разработчиков и пользователей, неэффективного управления разработкой в условиях изменяющихся требований, неконтролируемости изменений в процессе выполнения работ, субъективности в оценке качества продуктов разработки и т.д.

2. Причины возникновения ООП

На первых компьютерах для написания программ сначала применялись машинные коды, а затем язык ассемблера. Но этот язык не соответствует сегодняшним стандартам. По мере роста сложности программ оказалось, что разработчики не в состоянии помнить всю информацию, нужную для отладки и совершенствования их программ. Какие значения хранятся в регистрах? Есть ли уже переменная с этим именем? Какие переменные надо инициализировать, перед тем как передать управление следующему коду?

Частично эти проблемы решили первые языки высокого уровня: Фортран, Кобол, Алгол. Но рост сложности программ продолжался, и появились проекты, в которых ни один программист не мог удержать в голове все детали. Над проектами стали работать команды программистов. Значительная взаимозависимость частей ПО мешает создавать ПО по типу конструирования материальных объектов. Например, здание, автомобиль и электроприборы обычно собираются из готовых компонент, которые не надо разрабатывать «с нуля». Многократное использование ПО – цель, к которой постоянно стремятся, но и которой редко достигают. Из программной системы тяжело извлечь независимые фрагменты. ООП облегчает эту задачу.

3. Базовые понятия

Само по себе применение объектно-ориентированного языка не вынуждает к написанию объектно-ориентированных программ, хотя и упрощает их разработку. Чтобы эффективно использовать ООП, требуется рассматривать задачи иным способом, нежели это принято в процедурном программировании.

Известно утверждение, применимое к естественным языкам, что язык, на котором высказывается идея, направляет мышление. Как для компьютерных, так и для естественных языков справедливо: язык направляет мысли, но не предписывает их.

Аналогично, объектно-ориентированная техника не снабжает программиста новой вычислительной мощностью, которая бы позволила решить проблемы, недоступные для других средств. Но объектно-ориентированный подход делает задачу проще и приводит ее к более естественной форме. Это позволяет обращаться с проблемой таким образом, который благоприятствует управлению большими программными системами.

ООП часто называется новой парадигмой программирования. Парадигма программирования – способ концептуализации, который определяет, как проводить вычисления и как работа, выполняемая компьютером, должна быть структурирована и организована.

Процесс разбиения задачи на отдельные, структур-

но связанные, части, называется **декомпозицией**. При процедурной декомпозиции в задаче выделяются алгоритмы и обрабатываемые ими структуры данных, при логической – правила, связывающие отдельные понятия. При объектно-ориентированной декомпозиции в задаче выделяются классы и способы взаимодействия объектов этих классов друг с другом.

Главный способ борьбы со сложностью ПО – **абстрагирование**, т.е. способность отделить логический смысл фрагмента программы от проблемы его реализации.

Абстрагирование – это выделение таких существенных характеристик объекта, которые отличают его от всех других видов объектов и таким образом чётко определяются особенности данного объекта с точки зрения дальнейшего его рассмотрения. Абстрагирование позволяет отделить самые существенные особенности поведения от несущественных. Абстракция определяет существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и четко очерчивает концептуальную границу объекта с точки зрения наблюдателя.

Модульность – улучшенный метод создания и управления совокупностями имен и связанными с ними значениями. Суть модуля состоит в разбиении пространства имен на две части: открытая (public) часть является доступной извне модуля, закрытая (private) часть доступна только внутри модуля.

Иерархия – ранжированная (упорядоченная) система абстракций.

4. Обзор существующих методологий

4.1. Методология процедурно-ориентированного программирования

Появление первых электронных вычислительных машин или компьютеров ознаменовало новый этап в развитии техники вычислений. Казалось, достаточно разработать последовательность элементарных действий, каждое из которых преобразовать в понятные компьютеру инструкции, и любая вычислительная задача может быть решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ. Появились специальные языки программирования, которые позволили преобразовывать отдельные вычислительные операции в соответствующий программный код.

К таким языкам относятся Assembler, C, Pascal и другие.

Основой данной методологии разработки программ являлась процедурная или алгоритмическая организация структуры программного кода. Это было настолько естественно для решения вычислительных задач, что ни у кого не вызывала сомнений целесообразность такого подхода. Исходным понятием этой методологии являлось понятие **алгоритма**, под которым, в общем случае, понимается

некоторое предписание выполнить точно определенную последовательность действий, направленных на достижение заданной цели или решение поставленной задачи.

Принято считать, что сам термин алгоритм происходит от имени средневекового математика Аль-Хорезми, который в 825 г. описал правила выполнения арифметических действий в десятичной системе счисления.

Вся история математики тесно связана с разработкой тех или иных алгоритмов решения актуальных для своей эпохи задач. Более того, само понятие алгоритма стало предметом соответствующей теории – **теории алгоритмов**, которая занимается изучением общих свойств алгоритмов. Со временем содержание этой теории стало настолько абстрактным, что соответствующие результаты понимали только специалисты. Поэтому какое-то время языки программирования назывались алгоритмическими, а первое графическое средство документирования программ получило название **блок-схемы алгоритма**.

Потребности практики не всегда требовали установления вычислимости конкретных функций или разрешимости отдельных задач. В языках программирования возникло и закрепилось новое понятие **процедуры**, которое конкретизировало общее понятие алгоритма применительно к решению задач на компьютерах. Так же, как и алгоритм, процедура представляет собой законченную последовательность действий или операций, направленных на решение отдельной задачи. В языках программирования появилась специальная синтаксическая конструкция, которая получила название процедуры.

Со временем разработка больших программ преврати-

лась в серьезную проблему и потребовала их разбиение на более мелкие фрагменты. Основой для такого разбиения как раз и стала **процедурная декомпозиция**, при которой отдельные части программы или **модули** представляли собой совокупность процедур для решения некоторой совокупности задач. Главная особенность процедурного программирования заключается в том, что программа всегда имеет начало во времени или начальную процедуру и окончание. При этом вся программа может быть представлена визуально в виде направленной последовательности графических примитивов или блоков.

Графическое представление программы в виде последовательных процедур

Важным свойством таких программ является необходимость завершения всех действий предшествующей процедуры для начала действий последующей процедуры. Изменение порядка выполнения этих действие даже в пределах одной процедуры потребовало включения в языки программирования специальных условных операторов типа **if-else** и **goto** для реализации ветвления вычислительного процесса в зависимости от промежуточных результатов решения задачи.

Появление и интенсивное использование условных операторов и оператора безусловного перехода стало предметом острых дискуссий среди специалистов по программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода **goto** способно серьезно осложнить понимание кода. Соответствующие программы стали сравнивать со спагетти, имея в виду многочисленные переходы от одного фрагмента программы к другому, или, что еще хуже, возврат от конечных операторов программы к ее начальным операторам. Ситуация казалась настолько драматичной, что в литературе зазвучали призывы исключить оператор **goto** из языков программирования. Именно с тех пор принято считать хорошим стилем программирование – программирование без **goto**.

Рассмотренные идеи способствовали становлению некоторой системы взглядов на процесс разработки программ и написания программных кодов, которая получила название **методологии структурного программирования.** Основой данной методологии является процедурная декомпозиция программной системы и организация отдельных

модулей в виде совокупности выполняемых процедур. В рамках данной методологии получило развитие **нисходящее проектирование** программ или программирование «сверху-вниз». Период наибольшей популярности идей структурного программирования приходиться на конец 70-х – начало 80-х годов.

Как вспомогательное средство структуризации программного кода было рекомендовано использование отступов в начале каждой строки, которые должны выделять вложенные циклы и условные операторы. Все это призвано способствовать пониманию или читабельности самой программы.

Вот пример программы на языке С, иллюстрирующий эту особенность написания программ:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main()
{
  const int n=20;
  int Mas[n];
  srand(time(0));
  for(int i=0;i<n;i++)
  {
  Mas[i] = rand()%100;
  cout<<Mas[i]<<" ";
  }
  for(int i=0;i<n;i++)
  {
  bool sort = true;
  for(int k=0;k<n-1;k++)
  {
```

```
  if(Mas[k]>Mas[k+1])
  {
  int n = Mas[k];
  Mas[k] = Mas[k+1];
                Mas[k+1]=n;
  sort=false;
  }
  }
  if(sort==true) break;
  }
  cout<<endl<<endl;
  for(int i=0;i<n;i++)
  {
  cout<<Mas[i]<<" ";
  }
}
```

В этот период основным показателем сложности разработки программ считали ее размер. Вполне серьезно обсуждали такие оценки сложности программ, как количество строк программного кода. Правда, при этом делались некоторые предположения относительно синтаксиса самих строк, которые должны были удовлетворять определенным правилам. Общая трудоемкость разработки программ оценивалась специальной единицей измерения – «человеко-месяц» или «человеко-год». А профессионализм программиста напрямую связывался с количеством строк программного кода, который он мог написать и отладить в течение, скажем, месяца.

4.2. Методология объектно-ориентированного программирования

Со временем ситуация стала существенно изменяться. Оказалось, что трудоемкость разработки программных приложений на начальных этапах программирования оценивалась значительно ниже реально затрачиваемых усилий, что служило причиной дополнительных расходов и затягивания окончательных сроков готовности программ. В процессе разработки приложений изменялись функциональные требования заказчика, что еще более отдаляло момент окончания работы программистов. Увеличение размеров программ приводило к необходимости привлечения большего числа программистов, что, в свою очередь, потребовало дополнительных ресурсов для организации их согласованной работы.

Но не менее важными оказались качественные изменения, связанные со смещением акцента использования компьютеров. Если в эпоху «больших машин» основными потребителями программного обеспечения были крупные предприятия, компании и учреждения, то позже появились персональные компьютеры и стали повсеместным атрибутом мелкого и среднего бизнеса. Вычислительные и расчетно-алгоритмические задачи в этой области традиционно занимали второстепенное место, а на первый план выступили задачи обработки и манипулирования данными.

Стало очевидным, что традиционные методы процедурного программирования не способны справиться ни с растущей сложностью программ и их разработки, ни с

необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Такой методологией стало **объектно-ориентированное программирование (ООП)**.

Фундаментальными понятиями ООП являются понятия класса и объекта. При этом под классом понимают некоторую абстракцию совокупности объектов, которые имеют общий набор свойств и обладают одинаковым поведением. Каждый объект в этом случае рассматривается как экземпляр соответствующего класса. Объекты, которые не имеют полностью одинаковых свойств или не обладают одинаковым поведением, по определению, не могут быть отнесены к одному классу.

Основными принципами ООП являются наследование, инкапсуляция и полиморфизм.

Наследование – это принцип, в соответствии с которым знание о более общей категории разрешается применять для более узкой категории.

Инкапсуляция – это сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей. Инкапсуляция ведет свое происхождение от деления модуля в некоторых языках программирования на две части или секции: интерфейс и реализацию. Интерфейс содержит всю информацию, необходимую для взаимодействия с другими объектами. Реализация скрывает или маскирует от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

Полиморфизм – это способность некоторых объектов принимать различные внешние формы в зависимости от обстоятельств. Применительно к ООП полиморфизм означает, что действия, выполняемые одноименными методами, могут отличаться в зависимости от того, какому из классов относиться то или иной метод. Полиморфизм объектно-ориентированных языков связан с перегрузкой функций, но не тождествен ей. Важно иметь в виду, что имена методов и свойств тесно связаны с классами, в которых они описаны. Это обеспечивает определенную надежность работы программы, поскольку исключает случайное применение метода для решения несвойственных ему задач.

Появление объектно-ориентированных языков программирования было связано с необходимостью реализации концепции классов и объектов на синтаксическом уровне. Включение в известные языки программирования **C** и **Pascal** классов привело к появлению соответственно **C++** и **Object Pascal**.

Широкое распространение методологии ООП оказало влияние на процесс разработки программ. В частности, процедурно-ориентированная декомпозиция программ уступила место **объектно-ориентированной декомпозиции**, при которой отдельными структурными единицами программы стали являться не процедуры и функции, а классы и объекты с соответствующими свойствами и методами. Как следствие, программа перестала быть последовательностью предопределенных на этапе кодирования действий, а **стала событийно-управляемой**. Последнее обстоятельство стало доминирующим при разработке широкого круга современных приложений. В этом случае

каждая программа представляет собой бесконечный цикл ожидания некоторых заранее определенных событий. Инициаторами событий могут быть другие программы или пользователи.

Наиболее существенным обстоятельством в развитии методологии ООП явилось осознание того факта, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Действительно, до того как начать программирование классов, их свойств и методов, необходимо определить, чем же являются сами эти классы. Более того, нужно дать ответы на такие вопросы, как: сколько и какие классы нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания классам требуемого поведения, а также установить взаимосвязи между классами.

Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной предметной области, для которой разрабатывается программа. Все эти обстоятельства привели к появлению специальной **методологии объектно-ориентированного анализа и проектирования (ООАП)**.

4.3. Методология объектно-ориентированного анализа и проектирования

Необходимость анализа предметной области до начала написания программы была осознана давно при разработке масштабных проектов.

Выделение исходных или базовых компонентов предметной области, необходимых для решения той или иной задачи, представляет, в общем случае, нетривиальную проблему. Сложность данной проблемы проявляется в неформальном характере процедур или правил, которые можно применять для этой цели. Более того, такая работа должна выполняться совместно со специалистами или экспертами, хорошо знающими предметную область.

Для выделения или идентификации компонентов предметной области было предложено несколько способов и правил. Сам этот процесс получил название **концептуализации** предметной области. При этом под **компонентой** понимают некоторую абстрактную единицу, которая обладает функциональностью, т.е. может выполнять определенные действия, связанные с решением поставленных задач.

Появление методологии ООАП потребовало, с одной стороны, разработки различных средств концептуализации предметной области, а с другой – соответствующих специалистов, которые владели бы этой методологией. На данном этапе появляется относительно новый тип специалиста, который получил название **аналитика** или **архитектора**.

Наряду со специалистами по предметной области аналитик участвует в построении концептуальной схемы будущей программы, которая затем преобразуется программистами в код. При этом отдельные компоненты выбираются таким образом, чтобы при последующей разработке их было удобно представить в форме классов и объектов. В этом случае немаловажное значение

приобретает и сам язык представления информации о концептуальной схеме предметной области.

Разделение процесса разработки сложных программных приложений на отдельные этапы способствовало становлению концепции жизненного цикла программы. Под **жизненным циклом** (ЖЦ) программы понимают совокупность взаимосвязанных и следующих во времени этапов, начиная от разработки требований к ней и заканчивая полным отказом от ее использования.

Согласно принятым взглядам ЖЦ программы состоит из следующих этапов:

- анализа предметной области и формулировки требований к программе;
- проектирования структуры программы;
- реализации программы в кодах;
- внедрения программы;
- сопровождения программы;
- отказа от использования программы.

На этапе анализа предметной области формулировки требований осуществляется определение функций, которые должна выполнять разрабатываемая программа, а также концептуализация предметной области. Эту работу выполняют аналитики совместно со специалистами предметной области. Результатом данного этапа должна являться некоторая концептуальная схема, содержащая описание основных компонентов и тех функций, которые они должны выполнять.

Этап проектирования структуры программы заключается в разработке детальной схемы будущей программы, на которой указываются классы, их свойства и методы, а

также различные взаимосвязи между ними. Как правило, на этом этапе могут участвовать в работе аналитики, архитекторы и отдельные квалифицированные программисты. Результатом данного этапа должна стать детализированная схема программы, на которой указываются все классы и взаимосвязи между ними в процессе функционирования программы. Согласно методологии ООАП, именно данная схема должна служить исходной информацией для написания программного кода.

Результатом этапа программирования является программное приложение, которое обладает требуемой функциональностью и способно решать нужные задачи в конкретной предметной области.

Этапы внедрения и сопровождения программы связаны с необходимостью настройки и конфигурирования среды программирования, а также с устранением возникших в процессе ее использования ошибок.

Иногда в качестве отдельного этапа выделяют тестирование программы, под которым понимают проверку работоспособности программы на некоторой совокупности исходных данных или при некоторых специальных режимах эксплуатации.

Результатом этих этапов является повышение надежности программного приложения, исключающего возникновение критических ситуаций или нанесение ущерба компании, использующей данное приложение.

Методология ООАП тесно связана с концепцией автоматизированной разработки программного обеспечения **(Computer Aided Software Engineering, CASE)**. Появление первых **CASE**-средств было встречено с определенной нас-

тороженностью. Со временем появились как восторженные отзывы об их применении, так и критические оценки их возможностей. Причин для столь противоречивых мнений было несколько. Первая из них заключается в том, что ранние **CASE**-средства были простой надстройкой над некоторой системой управления базами данных. Хотя визуализация процесса разработки концептуальной схемы БД имеет немаловажное значение, она не решает проблем разработки приложений других типов.

Вторая причина имеет более сложную природу, поскольку связана с графической нотацией, реализованной в том или ином **CASE**-средстве. Если языки программирования имеют строгий синтаксис, то попытки предложить подходящий синтаксис для визуального представления концептуальных схем БД были восприняты далеко неоднозначно. На этом фоне появление унифицированного языка моделирования **(Unified Modeling Language, UML)**, который ориентирован на решение задач первых двух этапов ЖЦ программ, было воспринято с большим оптимизмом.

4.4. Методология системного анализа и системного моделирования

Системный анализ как научное направление имеет более давнюю историю, чем ООП и ООАП, и собственный предмет исследования. Центральным понятием системного анализа является понятие **системы**, под которой понимается совокупность объектов, компонентов или элементов произвольной природы, образующих некоторую целостность.

Определяющей предпосылкой выделения некоторой совокупности как системы является возникновение у нее новых свойств, которых не имеют составляющие ее элементы. Примеров систем можно привести достаточно много – это персональный компьютер, автомобиль, человек, биосфера, программа и др.

Важнейшими характеристиками любой системы являются ее структура и процесс функционирования. Под **структурой** системы понимают устойчивую во времени совокупность взаимосвязей между ее элементами или компонентами. Именно структура связывает воедино все элементы и препятствует распаду системы на отдельные компоненты. Структура системы может отражать самые различные взаимосвязи, в том числе и вложенность элементов одной системы в другую. В этом случае принято называть более мелкую или вложенную систему **подсистемой**, а более крупную – **метасистемой**.

Процесс функционирования системы тесно связан с изменением ее свойств или поведения во времени. При этом важной характеристикой системы является ее **состояние**, под которым понимается совокупность свойств или признаков, которые в каждый момент времени отражают наиболее существенные особенности поведения системы.

Процесс функционирования системы отражает поведение системы во времени и может быть представлен как последовательное изменение ее состояний. Если система изменяет одно свое состояние на другое, то принято говорить, что **система** переходит из одного состояния в другое. Совокупность признаков или условий изменения состояния системы в этом случае называется **переходом**.

Методология системного анализа служит концептуальной основой системно-ориентированной декомпозиции предметной области. В этом случае исходными компонентами концептуализации являются системы и взаимосвязи между ними. При этом понятие системы является более общим, чем понятие классов и объектов в ООАП. Результатом системного анализа является построение некоторой модели системы или предметной области.

Важность построения моделей заключается в возможности их использования для получения информации о свойствах или поведении системы-оригинала. При этом процесс построения и последующего применения моделей для получения информации о системе-оригинале получил название **моделирование**.

Наиболее общей моделью системы является так называемая модель «черного ящика». В этом случае система представляется в виде прямоугольника, внутреннее устройство которого скрыто от аналитика или неизвестно. Однако система не является полностью изолированной от внешней среды, поскольку последняя оказывает на систему некоторые информационные или материальные воздействия. Такие воздействия получили название **входных воздействий**. В свою очередь, система также оказывает на среду или другие системы определенные информационные или материальные воздействия, которые получили название выходных воздействий. Графически данная модель может быть изображена следующим образом:

Ценность моделей, подобных модели «черного ящика», весьма условна. Однако, общая модель системы содержит некоторую важную информацию о функциональных особенностях данной системы, которые дают представление о ее поведении. Действительно, кроме самой общей информации о том, на какие воздействия реагирует система, и как проявляется эта реакция на окружающие объекты и системы, другой информации мы получить не можем.

Процесс разработки адекватных моделей и их последующего конструктивного применения требует не только знания общей методологии системного анализа, но и наличия соответствующих изобразительных средств или языков для фиксации результатов моделирования и их документирования. Очевидно, что естественный язык не вполне подходит для этой цели, поскольку обладает неоднозначностью и неопределенностью. Для построения моделей были разработаны достаточно серьезные теоретические методы, основанные на развитии математических и логических средств моделирования, а также предложены различные формальные и графические нотации, отражающие специфику решаемых задач. Важно представлять, что унификация любого языка моделирования тесно связана с методологией системного моделирования.

Сложность системы и, соответственно, ее модели может быть рассмотрена с различных точек зрения. Прежде всего, можно выделить сложность структуры системы, которая характеризуется количеством элементов системы и различными типами взаимосвязей между этими элементами. Вторым аспектом сложности является сложность процесса функционирования системы. Это может быть связано как с непредсказуемым характером поведения системы, так и с невозможностью формального представления правил преобразования входных воздействий в выходные.

5. Классы и объекты

«Объект представляет собой особый опознаваемый предмет, блок или сущность (реальную или абстрактную), имеющую важное функциональное назначение в данной предметной области». **(Смитт, Токей)**

По определению будем называть объектом понятие, абстракцию или любую вещь с четко очерченными границами, имеющую смысл в контексте рассматриваемой прикладной проблемы. Введение объектов преследует две цели:

- понимание прикладной задачи (проблемы);
- введение основы для реализации на компьютере.

Объект обладает состоянием, поведением и индивидуальностью. Объекты существуют во времени, изменяются, имеют внутреннее состояние, преходящи и могут создаваться, разрушаться и разделяться.

Структура и поведение схожих объектов определяет общий для них класс.

Состояние объекта характеризуется перечнем всех возможных (обычно статических) свойств данного объекта и текущими значениями (обычно динамическими) каждого из этих свойств.

Поведение определяется последовательностью совершаемых над объектом действий.

Поведение характеризует то, как объект воздействует или подвергается воздействию других объектов с точки

зрения изменения состояния этих объектов и передачи сообщений.

Объединение объектов в классы позволяет ввести в задачу абстракцию и рассмотреть ее в более общей постановке. Класс имеет имя, которое относится ко всем объектам этого класса. Кроме того, в классе вводятся имена атрибутов, которые определены для объектов. В этом смысле описание класса аналогично описанию типа структуры (записи); при этом каждый объект имеет тот же смысл, что и экземпляр структуры (переменная или константа соответствующего типа).

Атрибут – это значение, характеризующее объект в его классе. Примеры атрибутов.

Среди атрибутов различаются постоянные атрибуты (константы) и переменные атрибуты. Постоянные атрибуты характеризуют объект в его классе (например, номер счета, категория, имя человека и т.п.). Текущие значения переменных атрибутов характеризуют текущее состояние объекта (например, баланс счета, возраст человека и т.п.); изменяя значения этих атрибутов, мы изменяем состояние объекта.

Операции над объектом

Операция – это функция (или преобразование), которую можно применять к объектам данного класса.

Все объекты данного класса используют один и тот же экземпляр каждой операции (т.е. увеличение количества объектов некоторого класса не приводит к увеличению количества загруженного программного кода). Объект,

из которого вызвана операция, передается ей в качестве ее неявного аргумента (параметра this).

Одна и та же операция может, вообще говоря, применяться к объектам разных классов: такая операция называется полиморфной, так как она может иметь разные формы для разных классов. Например, для объектов классов вектор и комплексное_число можно определить операцию +; эта операция будет полиморфной, так как сложение векторов и сложение комплексных чисел, вообще говоря, разные операции.

Каждой операции соответствует метод – реализация этой операции для объектов данного класса. Таким образом, операция – это спецификация метода, метод – реализация операции.

Операция (и реализующие ее методы) определяется своей сигнатурой, которая включает, помимо имени операции, типы (классы) всех ее аргументов и тип (класс) результата (возвращаемого значения). Все методы, реализующие операцию должны иметь такую же сигнатуру, что и реализуемая ими операция.

При моделировании системы полезно различать операции, имеющие побочные эффекты (эти эффекты выражаются в изменении значений атрибутов объекта, т.е. в изменении его состояния), и операции, которые выдают требуемое значение, не меняя состояния объекта. Эти последние операции называются запросами.

Значения некоторых атрибутов объекта могут быть доступны только операциям этого объекта.

Запросы без аргументов (за исключением неявного аргумента – объекта, к которому применяется операция) могут рассматриваться как производные атрибуты.

Виды наиболее распространенных операций

1. Модификатор – изменяет состояние объекта путем записи или доступа.
2. Селектор – дает доступ для определения состояния объекта без его изменения (операция чтения).
3. Итератор – организация доступа к частям объекта в определенной последовательности.
4. Конструктор – создание или инициализация объекта
5. Деструктор – разрушение объекта или освобождение занимаемой им памяти.

Отношения между объектами

Между объектами можно устанавливать зависимости по данным. Эти зависимости выражают связи или отношения между классами указанных объектов. Отношения двух любых объектов основываются на знаниях, которыми обладает они друг о друге: об опреациях, которые можно выполнять, и об ожидаемом поведении.

Типы отношений между объектами:

1. Связи.
2. Агрегация.

В отношении связь объекты могут выполнять различные роли.

Актер или активный объект **(actor)** – объект может воздействовать, но никогда не подвержен воздействию (активный объект).

Сервер или исполнитель **(server)** – подвергается управлению со стороны других объектов, но никогда не активен.

Агент или посредник **(agent, broker)** – выпоняет роль как актера так и сервера, как правило создается в интересах активного объекта.

Связь типа агрегация это отношение типа часть-целое **(A part-of)**. Позволяет идя от целого (агрегата) прийти к его частям (атрибутам). По смыслу, агрегация может означать физическое вхождение одного объекта в другой (состав или структуру объекта), но может быть и другая семантика, например, принадлежность одного объекта другому.

Класс

Класс – это множество объектов связанных общностью структуры и поведения.

Как правило, выделяют интерфейс (видимый всем внешний облик и набор поддерживаемых методов) и реализацию класса (защищенное от других внутреннее устройство).

Для задания класса необходимо указать имя этого класса, а затем перечислить его атрибуты и операции (или методы). С каждым объектом связана структура данных, полями которой являются атрибуты этого объекта и указатели функций (фрагментов кода), реализующих операции этого объекта (отметим, что указатели функций в результате оптимизации кода обычно заменяются на обращения к

этим функциям). Таким образом, объект – это некоторая структура данных, тип которой соответствует классу этого объекта.

Иногда в подклассе бывает необходимо переопределить операцию, определенную в одном из его суперклассов. Для этого операция, которая может быть получена из суперкласса в результате наследования, определяется и в подклассе; это ее повторное определение «заслоняет» ее определение в суперклассе, так что в подклассе применяется не унаследованная, а переопределенная в нем операция.

Напомним, что каждая операция определяется своей сигнатурой; следовательно, сигнатура переопределения операции должна совпадать с сигнатурой операции из суперкласса, которая переопределяется данной операцией.

Переопределение может преследовать одну из следующих целей:

- расширение: новая операция расширяет унаследованную операцию, учитывая влияние атрибутов подкласса;
- ограничение: новая операция ограничивается выполнением лишь части действий унаследованной операции, используя специфику объектов подкласса;
- оптимизация: использование специфики объектов подкласса позволяет упростить и ускорить соответствующий метод;
- удобство.

Отношения между классами реализованные в языках программирования:

- Ассоциация – смысловая или семантическая связь между классами. Отношения один-к-одному, один-к-многим, многие-к-многим.
- Обобщение и наследование позволяют выявить аналогии между различными классами объектов, определяют многоуровневую классификацию объектов. Так, в графических системах могут существовать классы, определяющие обрисовку различных геометрических фигур: точек, линий (прямых, дуг окружностей и кривых, определяемых сплайнами), многоугольников, кругов и т.п.

Способ реализации – делегирование, когда объекты рассматриваются в качестве прототипов (образцов), которые делегируют свое поведение другим объектам, ограничивая потребность в создании новых классов. Реализует отношение общности и ассоциативности. Выделяют простое и множественное наследование. Абстрактные классы (не имеющие реализации объектов). Самый общий класс – базовый класс.

- Агрегация – это зависимость между классом составных объектов и классами, представляющими компоненты этих объектов (отношение «целое»–«часть»).
- Отношение разновидность или отношение общности (классификация).
- Полиморфизм (отнесение к нескольким типам).
- Использование. В реализации класса может быть использован другой класс. Пример: линия – точка.

- Наполнение. Сборный класс. Однородные классы состоят из объектов одного класса и неоднородные разных. Класс, экземпляры, которого состоят из наборов других классов. Параметризованные классы.
- Метаклассы (класс классов) трактует класс как объект.

  Отношения между классами и объектами.

  Классификация – средство упорядочения знаний.

  Методы классификации.

  Классическое распределение по категориям. Объекты, обладающие данным свойством относятся к одному классу.

  Концептуальное объединение. Вначале формируется концептуальное описание, а затем распределение по категориям. (латентные признаки).

  Теория прототипов. Сходство с прототипом класса.

6. Экскурс в Диаграммы

Под структурным системным анализом принято понимать метод исследования системы, который начинается с наиболее общего описания с последующей детализацией представления отдельных аспектов ее поведения и функционирования. При этом общая модель системы строится в виде некоторой иерархической структуры, которая отражает различные уровни абстракции с ограниченным числом компонентов на каждом из уровней. Одним из главных принципов структурного системного анализа является выделение на каждом из уровней абстракции только наиболее существенных компонентов или элементов системы.

В рамках программной инженерии принято рассматривать три графические нотации, получивших название диаграмм: диаграммы «сущность-связь» **(Entity-Relationship Diagrams, ERD)**, диаграммы функционального моделирования **(Structured Analysis and Design Techique, SADT)** и диаграммы потоков данных **(Data Flow Diagrams, DFD)**.

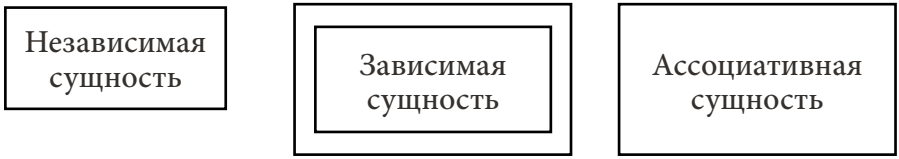
6.1. Диаграммы «сущность-связь»

Диаграммы «сущность-связь» предназначены для графического представления моделей данных разрабатываемой программной системы и предлагают некоторый набор стандартных обозначений для определения данных и отношений между ними. С помощью этого вида диаграмм

можно описать отдельные компоненты концептуальной модели данных и совокупность взаимосвязей между ними, имеющих важное значение для разрабатываемой системы.

Основными понятиями данной нотации являются понятия сущности и связи. При этом под сущностью **(entity)** понимается произвольное множество реальных или абстрактных объектов, каждый из которых обладает одинаковыми свойствами и характеристиками. В этом случае каждый рассматриваемый объект может являться **экземпляром** одной и только одной сущности, должен иметь уникальное имя или идентификатор, а также отличаться от других экземпляров данной сущности.

Примерами сущностей могут быть: банк, клиент банка, счет клиента, аэропорт, пассажир, рейс, компьютер, терминал, автомобиль, водитель. Каждая из сущностей может рассматриваться с различной степенью детализации и на различном уровне абстракции, что определяется конкретной постановкой задачи. Для графического представления сущностей используются специальные обозначения:



Связь (relationship) определяется как отношение или некоторая ассоциация между отдельными сущностями. Примерами связей могут являться родственные отношения типа «отец-сын» или производственные отношения типа «начальник-подчиненный». Другой тип связей за-

дается отношениями «иметь в собственности» или «обладать свойством» Различные типы связей графически изображаются в форме ромба с соответствующим именем данной связи.

Графическая модель данных строится таким образом, чтобы связи между отдельными сущностями отражали не только семантический характер соответствующего отношения, но и дополнительные аспекты обязательности связей, а также кратность участвующих в данных отношениях экземпляров сущностей.

Рассмотрим в качестве простого примера ситуацию, которая описывается двумя сущностями: «Сотрудник» и «Компания». При этом в качестве связи естественно использовать отношение принадлежности сотрудника данной компании. Если учесть соображения о том, что в компании работают несколько сотрудников, и эти сотрудники не могут быть работниками других компаний, то данный информация может быть представлена графически в виде следующей диаграммы «сущность-связь»:

На данном рисунке буква **«N»** около связи означает

тот факт, что в компании могут работать более одного сотрудника, при этом значение **N** заранее не фиксируется. Цифра **«1»** на другом конце связи означает, что сотрудник может работать только в одной конкретной компании, т.е. не допускается прием на работу сотрудников по совместительству из других компаний или учреждений.

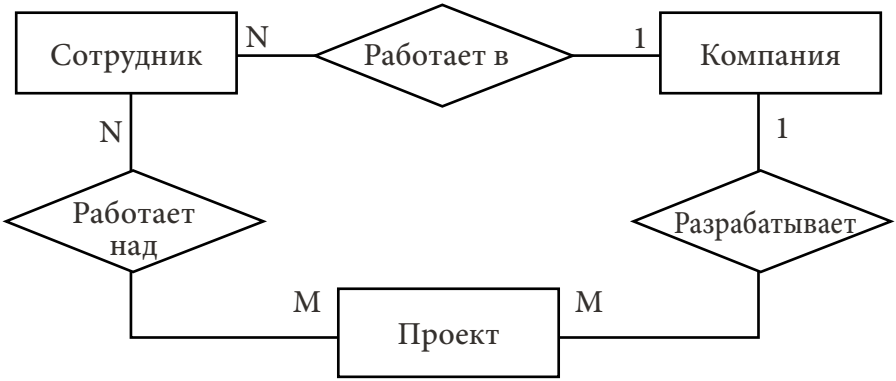
Немного другая ситуация складывается в случае рассмотрения сущностей «сотрудник» и «проект», и связи «участвует в работе над проектом».

Один сотрудник может участвовать в разработке нескольких проектов, а в разработке одного проекта могут принимать участие несколько сотрудников. Т.е. данная связь является многозначной. Данный факт специально отражается на диаграмме указанием букв «**N**» и «**M**» около соответствующих сущностей, при этом выбор конкретных букв не является принципиальным.

Рассмотренные две диаграммы могут быть объединены в одну, на которой будет представлена информация о сотрудниках компании, участвующих в разработке проектов данной компании. При этом может быть введена дополнительная связь, характеризующая проекты данной компании.

Ограниченность диаграмм «сущность-связь» проявляется при конкретизации концептуальной модели в более детальное представление моделируемой программной системы, которое кроме статических связей должно содер-

жать информацию о поведении или функционировании отдельных ее компонентов. Для этих целей используется другой тип диаграмм, получивших название диаграмм потоков данных.



6.2. Диаграммы функционального моделирования

Начало разработки диаграмм функционального моделирования относится к середине 1960-х годов, когда Дуглас Т. Росс предложил специальную технику моделирования, получившую название **SADT (Structured Analysis & Design Technique)**. Военно-воздушные силы США использовали методику **SADT** в качестве части своей программы интеграции компьютерных и промышленных технологий **(Integrated Computer Aided Manufacturing, ICAM)** и назвали ее **IDEF0 (Icam DEFinition)**. Целью программы **ICAM** было увеличение эффективности компьютерных технологий в сфере проектирования новых средств вооружений и ведения боевых действий. Одним из результатов этих исследований являлся вывод о том, что описательные языки не эффективны для документирования и моделирования процессов функционирования

сложных систем. Подобные описания на естественном языке не обеспечивают требуемого уровня непротиворечивости и полноты, имеющих доминирующее значение при решении задач моделирования.

В рамках программы **ICAM** было разработано несколько графических языков моделирования, которые получили следующие названия:

- Нотация **IDEF0** – для документирования процессов производства и отображения информации об использовании ресурсов на каждом из этапов проектирования систем.
- Нотация **IDEF1** – для документирования информации о производственном окружении систем.
- Нотация **IDEF2** – для документирования поведения системы во времени.
- Нотация **IDEF3** – специально для моделирования бизнес-процессов.

Нотация **IDEF2** никогда не была полностью реализована. Нотация **IDEF1** в 1985 году была расширена и переименована в **IDEFIX**. Методология **IDEF-SADT** нашла применение в правительственных и коммерческих организациях, поскольку на тот период времени вполне удовлетворяла различным требованиям, предъявляемым к моделированию широкого класса систем.

В начале 1990 года специально образованная группа пользователей **IDEF (IDEF User Group)**, в сотрудничестве с Национальным институтом по стандартизации и технологии США **(National Institutes for Standards and Technology, NIST)**, предприняла попытку создания стандарта для **IDEF0** и **IDEFIX**. Эта попытка оказалась

успешной и завершилась принятием в 1993 году стандарта правительства США известного как **FIPS** для данных двух технологий **IDEF0** и **IDEFIX**. В течение последующих лет этот стандарт продолжал активно развиваться и послужил основой для реализации в некоторых первых **CASE**-средствах.

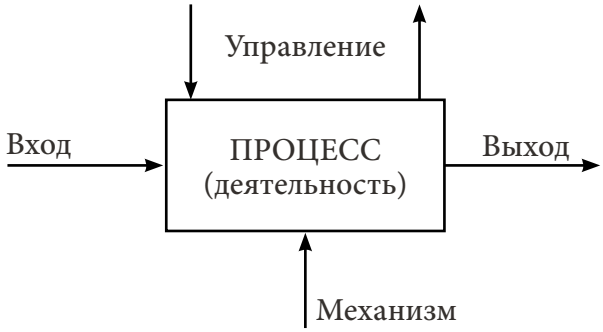
Методология **IDEF-SADT** представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели системы какой-либо предметной области. Функциональная модель **SADT** отображает структуру процессов функционирования системы и ее отдельных подсистем, т.е. выполняемы ими действия и связи между этими действиями. Для этой цели строятся специальные модели, которые позволяют в наглядной форме представить последовательность определенных действий. Исходными строительными блоками любой модели **IDEF0** процесса являются деятельность (activity) и стрелки (arrows).

Рассмотрим основные понятия методологии **IDEF-SADT**, которые используются при построении диаграмм функционального моделирования. Деятельность представляет собой некоторое действие или набор действий, которые имеют фиксированную цель и приводят к некоторому конечному результату. Иногда деятельность называют процессом. Модели **IDEF0** отслеживают различные виды деятельности системы, их описание и взаимодействие с другими процессами. На диаграммах деятельности или процесс изображается прямоугольником, который называется блоком. Стрелка служит для обозначения некоторого носителя или воздействия, которые обеспечивают перенос

данных или объектов от одной деятельности к другой. Стрелки также необходимы для описания того, что именно производит деятельность и какие ресурсы она потребляет. Это так называемые роли стрелок – **ICOM** – сокращение первых букв от названий соответствующих стрелок **IDEF0**. При этом различают стрелки четырех видов:

- **I (Input)** – вход, т.е. все, что поступает в процесс или потребляется процессом.
- **C (Control)** – управление или ограничения на выполнение операций процесса.
- **O (Output)** – выход или результат процесса.
- **M (Mechanism)** – механизм, который используется для выполнения процесса.

Методология **IDEF0** однозначно определяет, каким образом изображаются на диаграммах стрелки каждого вида **ICOM**. Стрелка Вход **(Input)** выходит из левой стороны рамки рабочего поля и входит слева в прямоугольник процесса. Стрелка Управление **(Control)** входит и выходит сверху. Стрелка Выход **(Output)** выходит из правой стороны процесса и входит в правую сторону рамки. Стрелка Механизм **(Mechanism)** входит в прямоугольник процесса снизу. Таким образом, базовое представление процесса на диаграммах IDEF0 имеет следующий вид:

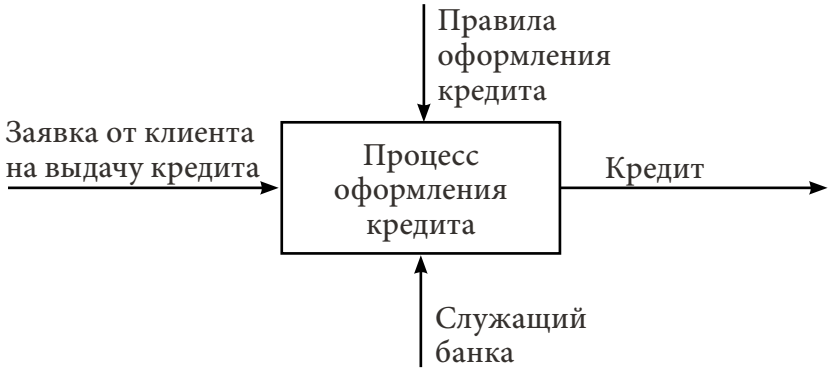


Техника построения диаграмм представляет собой главную особенность методологии **IDEF-SADT**. Место соединения стрелки с блоком определяет тип интерфейса. При этом все функции моделируемой системы и интерфейсы на диаграммах представляются в виде соответствующих блоков процессов и стрелок **ICOM**. Управляющая информация входи в блок сверху, в то время как информация, которая подвергается обработке, изображается с левой стороны блока. Результаты процесса представляются как выходы процесса и показываются с правой стороны блока. В качестве механизма может выступать человек или автоматизированная система, которые реализуют данную операцию. Соответствующий механизм на диаграмме представляется стрелкой, которая входит в блок процесса снизу.

Одной из наиболее важных особенностей методологии **IDEF-SADT** является постепенное введение все более детальных представлений модели системы по мере разработки отдельных диаграмм. Построение модели **IDEF-SADT** начинается с представления всей системы в виде простейшей диаграммы, состоящей из одного блока процесса и стрелок **ICOM**, служащих для изображения основных видов взаимодействия с объектами вне системы. Поскольку исходный процесс представляет всю систему как единое целое, данное представление является наиболее общим и подлежит дальнейшей декомпозиции.

Для иллюстрации основных идей методологии **IDEF-SADT** рассмотрим следующий простой пример. В качестве процесса будем представлять деятельность по оформлению кредита в банке. Входом данного процесса

является заявка от клиента на получение кредита, а выходом – соответствующий результат, т.е. непосредственно кредит. При этом управляющими факторами являются правила оформления кредита, которые регламентируют условия получения соответствующих финансовых средств в кредит. Механизмом данного процесса является служащий банка, который уполномочен выполнить все операции по оформлению кредита.



В конечном итоге модель **IDEF-SADT** представляет собой серию иерархически взаимосвязанных диаграмм с сопроводительной документацией, которая разбивает исходное представление сложной системы на отдельные составные части. Детали каждого из основных процессов представляются в виде более детальных процессов на других диаграммах. В этом случае каждая диаграмма нижнего уровня является декомпозицией некоторого процесса из более общей диаграммы. Поэтому на каждом шаге декомпозиции более общая диаграмма конкретизируется на ряд более детальных диаграмм.

В настоящее время диаграммы структурного систем-

ного анализа **IDEF-SADT** продолжают использоваться целым рядом организаций для построения и детального анализа функциональной модели существующих на предприятии бизнес-процессов, а также для разработки новых бизнес-процессов. Основной недостаток данной методологии связан с отсутствием явных средств для объектно-ориентированного представления моделей сложных систем. Хотя некоторые аналитики отмечают важность знания и применения нотации **IDEF-SADT**, ограниченные возможности этой методологии применительно к реализации соответствующих графических моделей в объектно-ориентированном программном коде существенно сужают диапазон решаемых с ее помощью задач.

6.3. Диаграммы потоков данных

Основой данной методологии графического моделирования информационных систем является специальная технология построения диаграмм потоков данных **DFD**.

Модель системы в контексте **DFD** представляется в виде некоторой информационной модели, основными компонентами которой являются различные потоки данных, которые переносят информацию от одной подсистемы к другой. Каждая из подсистем выполняет определенные преобразования входного потока данных и передает результаты обработки информации в виде потоков данных для других подсистем.

Основными компонентами диаграмм потоков данных являются:

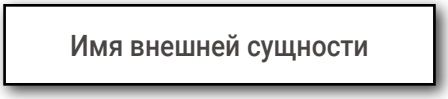
- внешние сущности;

- процессы;
- системы/подсистемы;
- накопители данных или хранилища;
- потоки данных.

Внешняя сущность представляет собой материальный объект или физическое лицо, которые могут выступать в качестве источника или приемника информации. Определение некоторого объекта или системы в качестве внешней сущности не являются строго фиксированным. Хотя внешняя сущность находится за пределами границ рассматриваемой системы, в процессе дальнейшего анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы модели системы. С другой стороны, отдельные процессы могут быть вынесены за пределы диаграммы и представлены как внешние сущности.

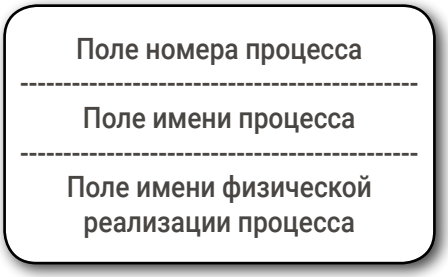
Примерами внешних сущностей могут служить: клиенты организации, заказчики, персонал, поставщики.

Внешняя сущность обозначается прямоугольником с тенью, внутри которого указывается ее имя. При этом в качестве имени рекомендуется использовать существительное в именительном падеже. Иногда внешнюю сущность называют также терминатором.



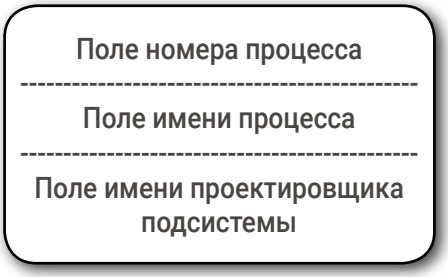
Процесс представляет собой совокупность операций по преобразованию входных потоков данных в выходные в соответствии с определенным алгоритмом или прави-

лом. Хотя физически процесс может быть реализован различными способами, наиболее часто подразумевается программная реализация процесса. Процесс на диаграмме потоков данных изображается прямоугольником с закругленными вершинами, разделенным на три секции или поля горизонтальными линиями.



Поле номера процесса служит для идентификации последнего. В среднем поле указывается имя процесса. В качестве имени рекомендовано использовать глагол в неопределенной форме с необходимыми дополнениями. Нижнее поле содержит указание на способ физической реализации процесса.

Информационная модель системы строиться как некоторая иерархическая схема в виде так называемой контекстной диаграммы, на которой исходная модель последовательно представляется в виде модели подсистем соответствующих процессов преобразования данных. При этом подсистема или система на контекстной диаграмме DFD изображается так же, как и процесс – прямоугольник с закругленными вершинами.



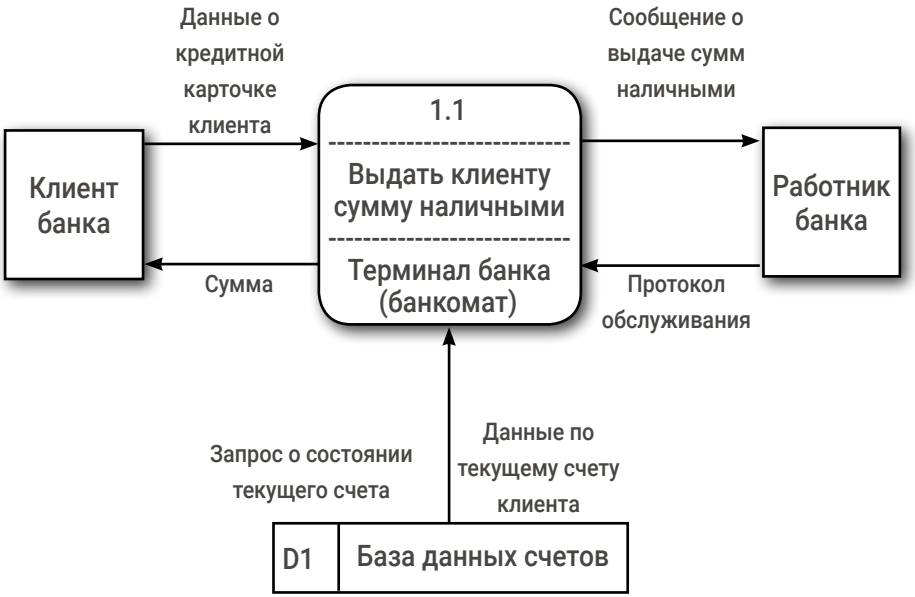
Накопитель данных или хранилище представляет собой абстрактное устройство или способ хранения информации, перемещаемой между процессами. Предполагается, что данные можно в любой момент поместить в накопитель и через некоторое время извлечь, причем физические способы помещения и извлечения данных могут быть произвольными. Накопитель данных может быть физически реализован различными способами, но наиболее часто предполагается его реализация в электронном виде на магнитных носителях. Накопитель данных на диаграмме потоков данных изображается прямоугольником с двумя полями. Первое служит для указания номера или идентификатора накопителя, который начинается с буквы «**D**». Второе поле служит для указания имени. При этом в качестве имени накопителя рекомендуется использовать существительное, которое характеризует способ хранения соответствующей информации.

Поле номера	Поле имени накопителя

Наконец, поток данных определяет качественный характер информации, передаваемой через некоторое соединение от источника к приемнику. Реальный поток данных может передаваться по сети между двумя компьютерами или любым другим способом, допускающим извлечение данных и их восстановление в требуемом формате. Поток данных на диаграмме DFD изображается линией со стрелкой на одном из ее концов, при этом стрелка показывает направление потока данных. Каждый поток данных имеет свое собственное имя, отражающее его содержание.

Таким образом, информационная модель системы в нотации DFD строится в виде диаграмм потоков данных, которые графически представляются с использованием соответствующей систем обозначений. В качестве примера рассмотрим упрощенную модель получения некоторой суммы наличными по кредитной карточке клиентом банка. Внешними сущностями данного примера являются клиент банка и, возможно, служащий банка, который контролирует процесс обслуживания клиентов. Накопителем данных может быть база данных о состоянии счетов отдельных клиентов банка. Отдельные потоки данных отражают характер передаваемой информации, необходимой для обслуживания клиента банка.

Соответствующая модель для данного пример может быть представлена следующим образом:



В настоящее время диаграммы потоков данных используются в некоторых **CASE** – средствах для построения информационных моделей систем обработки данных. Основной недостаток этой методологии также связан с отсутствием явных средств для объектно-ориентированного представления моделей сложных систем, а также для представления сложных алгоритмов обработки данных. Поскольку на диаграммах **DFD** не указываются характеристики времени выполнения отдельных процессов и передачи данных между процессами, то модели систем, реализующих синхронную обработку данных, не могут быть адекватно представлены в нотации **DFD**. Все эти особенности методологии структурного системного анализа ограничили возможности ее широкого применения и послужили основой для включения соответствующих средств в унифицированный язык моделирования.

7. История развития языка UML

Отдельные языки объектно-ориентированного моделирования стали появляться в период между серединой 1970-х и концом 1980-х годов, когда различные исследователи и программисты предлагали свои подходы к ООАП (методология объектно-ориентированного анализа и проектирования). В период между 1989–1994 гг. общее число наиболее известных языков моделирования возросло с 10 до более чем 50. Многие пользователи испытывали серьезные затруднения при выборе языка ООАП, поскольку ни один из них не удовлетворял всем требованиям, предъявляемым к построению моделей сложных систем. Принятие отдельных методик и графических нотаций в качестве стандартов **(IDEF0, IDEFIX)** не смогло изменить сложившуюся ситуацию непримиримой конкуренции между ними в начале 90-х годов.

К середине 1990-х некоторые из методов были существенно улучшены и приобрели самостоятельное значение при решении различных задач ООАП. Наиболее известными в этот период становятся:

- Метод Гради Буча **(Grady Booch)**, получивший условное название Booch.
- Метод Джеймса Румбаха **(James Rumbaugh)**, получивший название **Object Modeling Technique – OMT**.
- Метод Айвара Джекобсона **(Ivar Jacobson)**, получивший название **Object-Oriented Software Engineering – OOSE**.

Каждый из этих методов был ориентирован на поддержку отдельных этапов ООАП. Например, метод **OOSE** содержал средства представления вариантов использования, которые имеют существенное значение на этапе анализа требований в процессе проектирования бизнес-приложений. Метод **OMT** наиболее подходил для анализа процессов обработки данных в информационных системах. Метод **Booch** нашел наибольшее применение на этапах проектирования и разработки различных программных систем.

История развития языка UML берет начало с октября 1994 года, года Гради Буч и Джеймс Румбах из Rational Software Corporation начали работу по унификации методов Booch и OMT. Хотя сами по себе эти методы были достаточно популярны, совместная работа была направлена на изучение всех известных объектно-ориентированных методов с целью объединения их достоинств. При этом Г. Буч и Дж. Румбах сосредоточили усилия на полной унификации результатов своей работы. Проект так называемого унифицированного метода (Unified Method) был подготовлен и опубликован в октябре 1995 года. Осенью того же года к ним присоединился А. Джекобсон, главный технолог из компании Objectory AB (Швеция), с целью интеграции своего метода OOSE с двумя предыдущими.

Вначале авторы методов Booch, OMT и OOSE предполагали разработать унифицированный язык моделирования только для этих методик. С одной стороны, каждый из этих методов был проверен на практике и показал свою конструктивность при решении отдельных задач ООАП. Это давало основание для дальнейшей их модификации на основе устранения имеющегося несоответствия отдельных

понятий и обозначений. С другой стороны, унификация семантики и нотации должна была обеспечить некоторую стабильность на рынке объектно-ориентированных CASE-средств, которая необходима для успешного продвижения соответствующих программных инструментариев. Наконец, совместная работа давала надежду на существенное улучшение всех трех методов.

Начиная работу по унификации своих методов, Г. Буч, Дж. Румбах и А. Джекобсон сформулировали следующие требования к языку моделирования. Он должен:

- позволять моделировать не только программное обеспечение, но и более широкие классы систем и бизнес-приложений, с использованием объектно-ориентированных понятий;
- явным образом обеспечивать взаимосвязь между базовыми понятиями для моделей концептуального и физического уровней;
- обеспечивать масштабируемость моделей, что является важной особенностью сложных многоцелевых систем;
- быть понятен аналитикам и программистам, а также должен поддерживаться специальными инструментальными средствами, реализованными на различных компьютерных платформах.

Разработка системы обозначений или нотации для ООАП оказалась непохожей на разработку нового языка программирования. Во-первых, необходимо было решить две проблемы:

- должна ли данная нотация включать в себя спецификацию требований?

- следует ли расширять данную нотацию до уровня языка визуального программирования?

Во-вторых, было необходимо найти удачный баланс между выразительностью и простотой языка. С одной стороны, слишком простая нотация ограничивает круг потенциальных проблем, которые могут быть решены с помощью соответствующей системы обозначений. С другой стороны, слишком сложная нотация создает дополнительные трудности для ее изучения и применения аналитиками и программистами. В случае унификации существующих методов необходимо учитывать интересы специалистов, которые уже имеют опыт работы с ними, поскольку внесение серьезных изменений в новую нотацию может повлечь за собой непонимание и неприятие ее пользователями прежних методик. Чтобы исключить неявное сопротивление со стороны отдельных специалистов, необходимо учитывать интересы самого широкого круга пользователей. Последующая работа над языком UML должна была учесть все эти обстоятельства.

В этот период поддержка разработки языка UML становиться одной из целей консорциума OMG (Object Management Group). Хотя консорциум OMG образован еще в 1989 году с целью разработки предложений по стандартизации объектных и компонентных технологий CORBA, язык UML приобрел статус второго стратегического направления в работе OMG. Именно в рамках OMG создается команда разработчиков под руководством Ричарда Соли, которая будет обеспечивать дальнейшую работу по унификации и стандартизации языка UML. В июне 1995 года OMG организовала совещание всех круп-

ных специалистов и представителей входящих в консорциум компаний по методологии ООАП, на котором впервые в международном масштабе была признана целесообразность поиска индустриальных стандартов в области языков моделирования под эгидой OMG.

Усилия Г. Буча, Дж. Румбаха и А. Джекобсона привели к появлению первых документов содержащих описание собственно языка UML версии 0.9 (июнь 1996 г.) и версии 0.91 (октябрь 1996 г.). Имевшие статус запроса предложений RFP (Request For Proposal), эти документы послужили своеобразным катализатором для широкого обсуждения языка UML различными категориями специалистов. Первые отзывы и реакция на язык UML указывали на необходимость его дополнения отдельными понятиями и конструкциями.

В то же время стало ясно, что некоторые компании и организации видят в языке UML линию стратегических интересов для своего бизнеса. Компания Rational Software вместе с несколькими организациями, изъявившими желание выделить ресурсы для разработки строгого определения версии 1.0 языка UML, учредила консорциум партнеров UML. Компании, вошедшие в этот консорциум, обеспечили поддержку последующей работы по более точному и строгому определению нотации, что привело к появлению версии 1.0 языка UML. В январе 1997 года был опубликован документ с описанием языка UML 1.0, как начальный вариант ответа на запрос предложений RFP. Эта версия языка моделирования была достаточно хорошо определена, обеспечивала требуемую выразительность и мощность и предполагала решение широкого класса задач.

Инструментальные CASE-средства и диапазон их прак-

тического применения в большей степени зависят от удачного определения семантики и нотации соответствующего языка моделирования. Специфика языка UML заключается в том, что он определяет семантическую метамодель, а не модель конкретного интерфейса и способы представления или реализации компонентов.

В январе 1997 года целый ряд других компаний представили на рассмотрение OMG свои собственные ответы на запрос предложений RFP. В дальнейшем эти компании присоединились к партнерам UML, предлагая включить в язык UML некоторые свои идеи. В результате совместной работы с партнерами UML была предложена пересмотренная версия 1.1 языка UML. Особое внимание при разработке языка UML 1.1 было уделено достижению большей ясности семантики языка по сравнению с UML 1.0, а также учету предложений новых партнеров. Эта версия языка была представлена на рассмотрение OMG и была одобрена и принята в качестве стандарта OMG в ноябре 1997 года.

Очередной этап развития данного языка закончился в марте 1999 года, когда консорциумом OMG было опубликовано описание языка UML 1.3.

Статус языка UML определен как открытый для всех предложений по его доработке и совершенствованию. Сам язык UML не является чьей-либо собственностью и не запатентован кем-либо. В то же время аббревиатура UML является торговой маркой их законных владельцев.

Язык UML ориентирован для применения в качестве языка моделирования различными пользователями и научными сообществами для решения широкого класса задач ООАП.