**3 Проектирование и архитектура программного обеспечения**

**3.1 История развития технологии программирования и построения архитектуры ПО**

Технология программирования играла разную роль на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и методологии программирования росла и сложность решаемых на компьютерах задач, это привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на компьютерных носителях привело к широкому внедрению компьютеров практически во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно глубокое понятие качества ПС, причем предпочтение стало отдаваться не столько эффективности ПС, сколько удобству работы с ним для пользователей (не говоря уже о его надежности). Широкое использование компьютерных сетей привело к интенсивному развитию распределенных вычислений, дистанционного доступа к информации и электронного способа обмена сообщениями между людьми. Компьютерная техника из средства решения отдельных задач превращается в средство информационного моделирования реального и виртуального мира, способное просто отвечать людям на интересующие их вопросы.

Краткая характеристика, развития программирования по десятилетиям.

В **50-е годы** мощность компьютеров (первого поколения) была невелика, программирование велось, в основном, в машинном коде. Решались, главным образом, научно-технические задачи (счет по формулам), задание на программирование содержало достаточно точную постановку задачи. Использовалась интуитивная технология программирования. Программирование фактически было искусством, почти сразу приступали к составлению программы по заданию. Если задание изменялось, то время составления программы сильно увеличивалось, минимальная документация оформлялась уже после того, как программа начинала работать. Первое ПО имело простейшую структуру. Оно состояли из программы на машинном языке и обрабатываемых ею данных (рисунок 3.1). Сложность программ в машинных кодах ограничивалась способностью программиста одновременно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

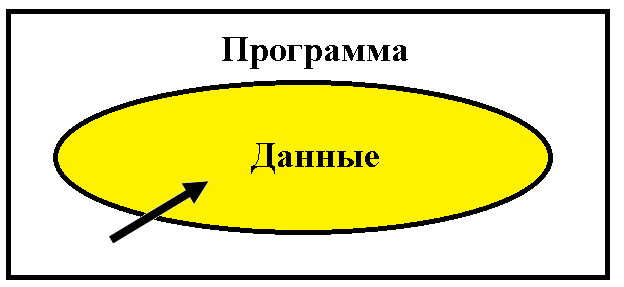


Рисунок 3.1 – Структура первых программ

Появление ассемблеров позволило вместо двоичных или 16-ричных кодов использовать символические имена данных и мнемоники кодов операций. В результате программы стали более "читаемыми".

Тем не менее, именно в этот период родилась фундаментальная для технологии программирования концепция модульного программирования, ориентированная на преодоления трудностей программирования в машинном коде. Появились первые языки программирования высокого уровня, из которых только ФОРТРАН пробился для использования в следующие десятилетия.

**60-е годы** характеризуются бурным развитием и широким использованием языков программирования высокого уровня (АЛГОЛ 60, ФОРТРАН, КОВОЛ ПЛ1и др.), значение которых в технологии программирования явно преувеличивалась. Надежда на то, что языки решат все проблемы, возникающие в процессе разработки больших программ, не оправдалась.

Революционным было появление в языках средств, позволяющих оперировать подпрограммами. Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рисунок 3.2).

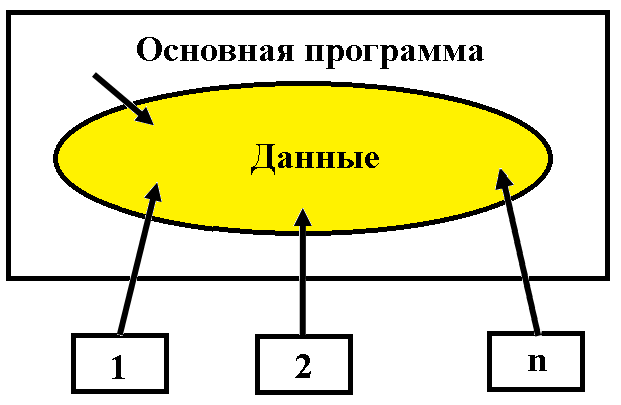


Рисунок 3.2 – Архитектура программы с глобальной областью данных

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок в подпрограммах, было предложено размещать локальные данные (рисунок 3.3).

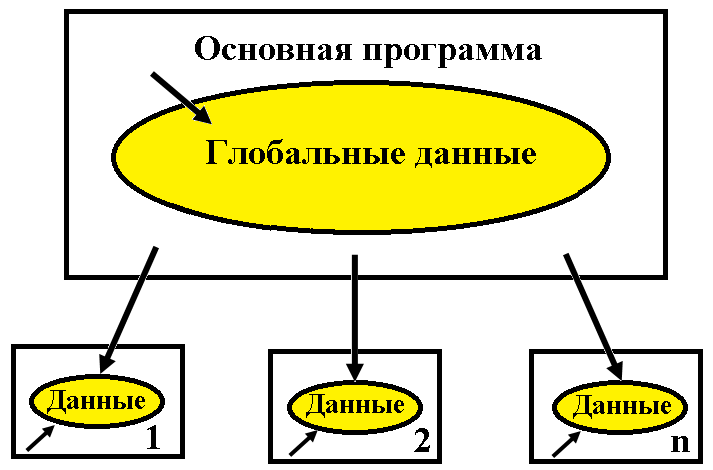


Рисунок 3.3 – Архитектура программы, использующей подпрограммы с локальными данными

Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне.

В результате повышения мощности компьютеров и накопления опыта программирования на языках высокого уровня быстро росла сложность решаемых задач, в результате чего обнаружилась ограниченность языков, проигнорировавших модульную организацию программ. В этот период стало понято, что важно не только то, на каком языке программируют, но и то, как программируют. Это было началом серьезных размышлений над методологией и технологией программирования. Появление компьютеров 2-го поколения привело к развитию мультипрограммирования и созданию относительно больших программных систем.

**В 70-е годы** получили широкое распространение информационные системы и базы данных. К середине 70-ых годов стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на традиционных. Это резко повысило интерес к компьютерным системам хранения данных. Началось интенсивное развитие технологии программирования, прежде всего, в следующих направлениях:

* обоснование и широкое внедрение нисходящей разработки и структурного программирования;
* развитие абстрактных типов данных и модульного программирования (в частности, возникновение идеи разделения спецификации и реализации модулей и использование модулей, скрывающих структуры данных);
* исследование проблем обеспечения надежности и мобильности программных средств;
* создание методики управления коллективной разработкой ПС;
* появление инструментальных программных средств (программных инструментов) поддержки технологии программирования.

Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит *декомпозиция* (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название *процедурной* декомпозиции.

В отличие от используемого ранее процедурного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось "сверху – вниз" и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм.

*Модульное программирование* предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые модули (библиотеки подпрограмм), например модуль графических ресурсов, модуль подпрограмм вывода на принтер (рисунок 3.4). Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым "внутренним" переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и С (C++), языки Ада и Modula.

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

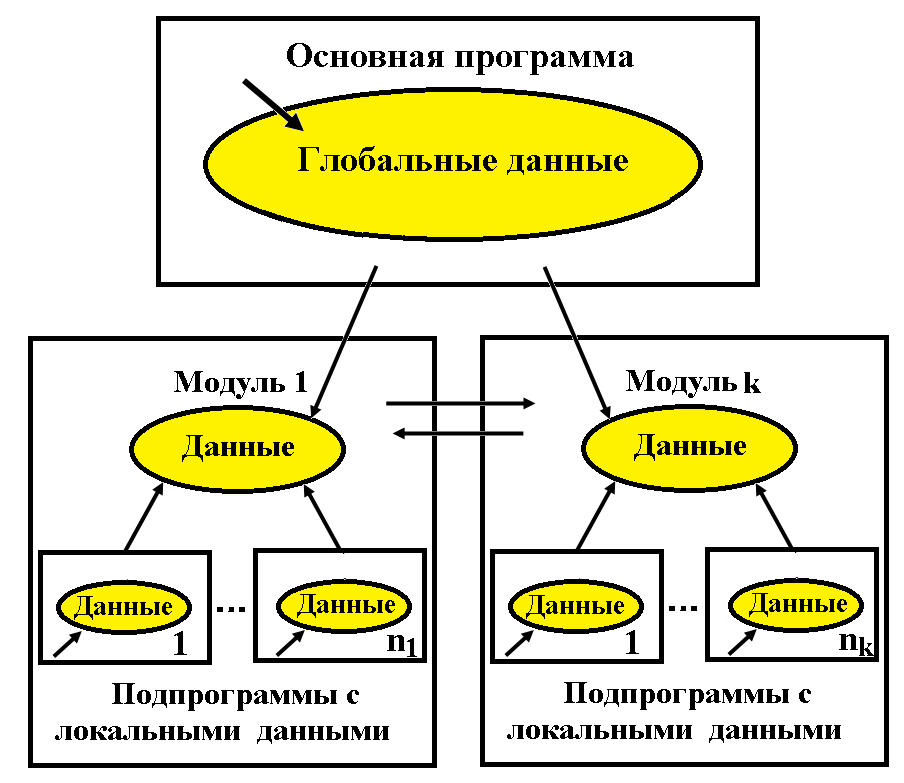


Рисунок 3.4 – Архитектура программы, состоящей из модулей

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема было предложено использовать объектный подход.

**80-е годы** характеризуются широким внедрением персональных компьютеров во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС. Это привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества ПС. На передовые позиции выходит объектный подход к разработке ПС. Создаются различные инструментальные среды разработки и сопровождения ПС. Развивается концепция компьютерных сетей.

Объектно-ориентированное программирование определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений (рисунок 3.5).

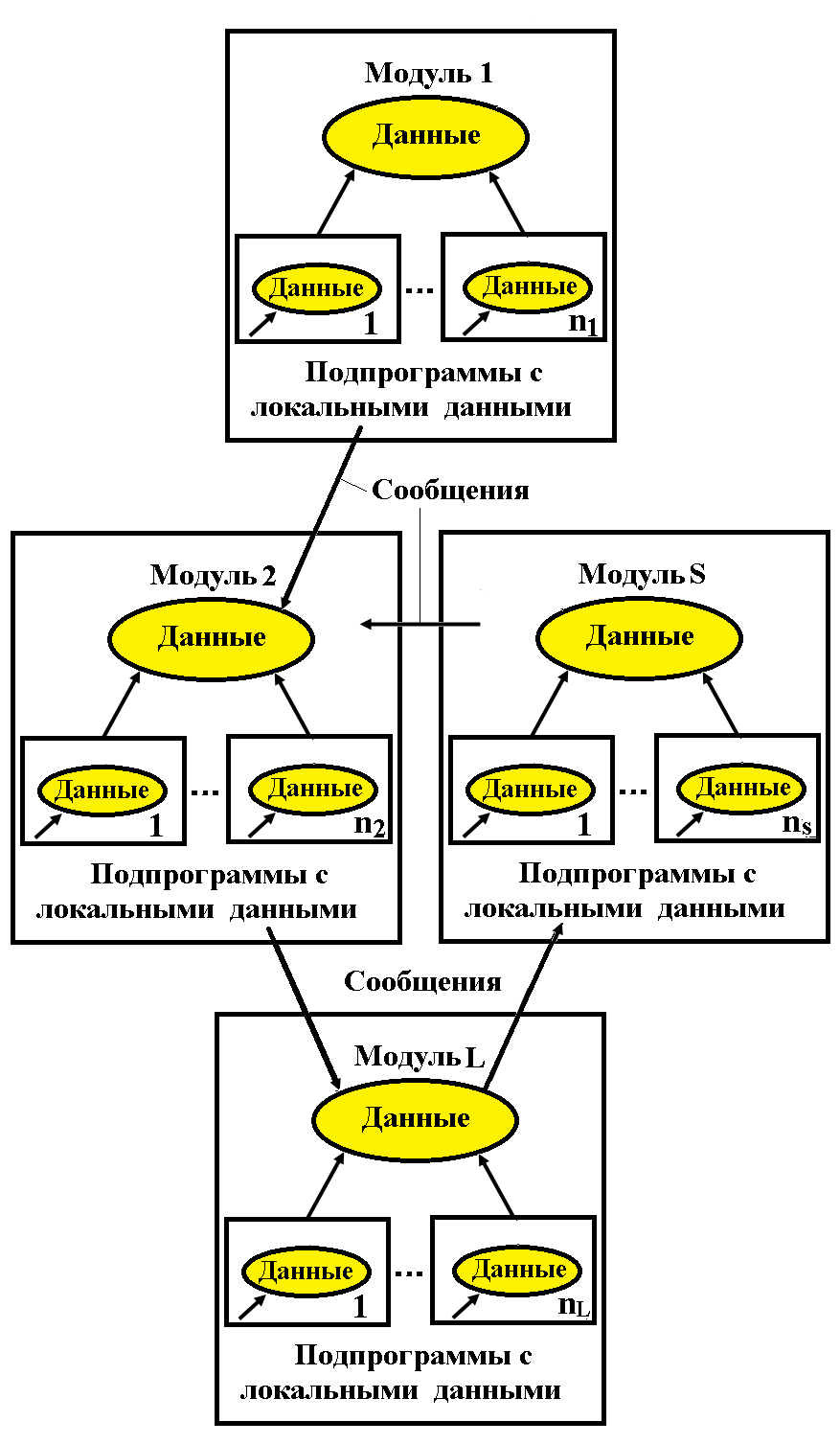


Рисунок 3.5 – Архитектура программы при объектно-ориентированном подходе

Объектная структура программы впервые была использована в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х годах XX в. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования Smalltalk (70-е годы XX в.), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, C++, Modula, Java.

Основным достоинством объектно-ориентированного программирования по сравнению со структурным является "более естественная" декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы.

Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы позволяют конструировать из сравнительно простых объектов сложные. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

Бурное развитие технологий программирования, основанных на объектном подходе, позволило решить многие проблемы. Так, были созданы среды, поддерживающие визуальное программирование, например Delphi, C#, Builder, Visual C++ и т. д. При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов. Результатом визуального проектирования является заготовка будущей программы, в которую уже внесены соответствующие коды.

**90-е годы** знаменательны широким охватом всего человеческого общества международной компьютерной сетью, персональные компьютеры стали подключаться к ней как терминалы. Это поставило ряд проблем (как технологического, так юридического и этического характера) регулирования доступа к информации компьютерных сетей. Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Стали бурно развиваться компьютерная технология (CASE-технология) разработки ПС и связанные с ней формальные методы спецификации программ. Начался решающий этап полной информатизации и компьютеризации общества.

Получили распространение новые направления разработки ПО.

**Компонентный подход**

Компонентный подход предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture – общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология СОМ фирмы Microsoft является развитием технологии OLE (Object Linking and Embedding – связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология СОМ определяет общую парадигму взаимодействия программ любых типов: библиотек, приложений, операционной системы, т. е. позволяет одной части программного обеспечения использовать функции (службы), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рисунок 3.6). Модификация СОМ, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM – распределенная СОМ).

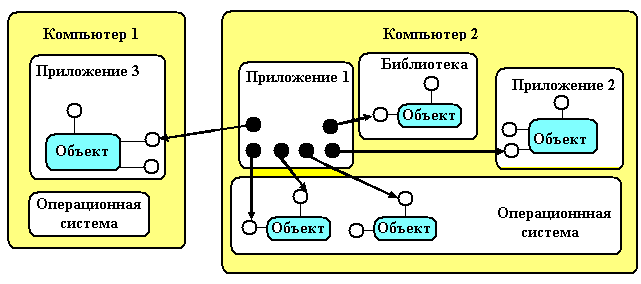


Рисунок 3.6 – Взаимодействие программных компонентов различных типов

OLE-automation или просто Automation (автоматизация) – технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений. Вводит понятие диспинтерфейса (dispinterface) – специального интерфейса, облегчающего вызов функций объекта. Эту технологию поддерживает, например, Microsoft Excel, предоставляя другим приложениям свои службы.

ActiveX – технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения как сосредоточенного на одном компьютере, так и распределенного в сети. Предполагает использование визуального программирования для создания компонентов – элементов управления ActiveX. Полученные таким образом элементы управления можно устанавливать на компьютер дистанционно с удаленного сервера, причем устанавливаемый код зависит от используемой операционной системы. Это позволяет применять элементы управления ActiveX в клиентских частях приложений Интернета.

Основными преимуществами технологии ActiveX, обеспечивающими ей широкое распространение, являются:

1) быстрое написание программного кода, поскольку все действия, связанные с организацией взаимодействия сервера и клиента, берет на себя программное обеспечение СОМ, программирование сетевых приложений становится похожим на программирование для отдельного компьютера;

2) открытость и мобильность – спецификации технологии были переданы в Open Group как основа открытого стандарта;

3) возможность написания приложений с использованием знакомых средств разработки, например Visual Basic, Visual C++, Borland Delphi, Borland C++ и любых средств разработки на Java;

4) большое количество уже существующих бесплатных программных элементов ActiveX (к тому же практически любой программный компонент OLE совместим с технологиями ActiveX и может применяться без модификаций в сетевых приложениях);

5) стандартность – технология ActiveX основана на широко используемых стандартах Интернет (TCP/IP, HTML, Java), с одной стороны, и стандартах, введенных в свое время Microsoft и необходимых для сохранения совместимости (COM, OLE).

MTS (Microsoft Transaction Server – сервер управления транзакциями) – технология, обеспечивающая безопасность и стабильную работу распределенных приложений при больших объемах передаваемых данных.

MIDAS (Multitier Distributed Application Server – сервер многозвенных распределенных приложений) – технология, организующая доступ к данным разных компьютеров с учетом балансировки нагрузки сети.

Все указанные технологии реализуют компонентный подход, заложенный в СОМ. Так, с точки зрения СОМ элемент управления ActiveX – внутренний сервер, поддерживающий технологию OLE-automation. Для программиста же элемент ActiveX – "черный ящик", обладающий свойствами, методами и событиями, который можно использовать как строительный блок при создании приложений.

**CASE-технологии**

Отличительной особенностью современного этапа развития технологии программирования, кроме изменения подхода, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые были названы CASE-технологиями (Computer-Aided Software/System Engineering – разработка программного обеспечения с использованием компьютерной поддержки). Без средств автоматизации разработка сложного программного обеспечения становится очень трудоемкой, т.к. память человека уже не в состоянии фиксировать все детали, которые необходимо учитывать при разработке программного обеспечения. Существуют CASE-технологии, поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию.

Появление нового подхода не означает, что отныне все программное обеспечение будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного программного обеспечения показывает, что он будет применяться достаточно широко.

# 

# 3.2 Основы проектирования программных систем

Известно, что технологический цикл конструирования программной системы (ПС) включает три процесса – анализ, синтез и сопровождение.

В ходе анализа ищется ответ на вопрос: "Что должна делать будущая ПС?". Именно на этой стадии закладывается фундамент успеха всего проекта. В процессе синтеза формируется ответ на вопрос: "Каким образом система будет реализовывать предъявляемые к ней требования?". Выделяют три этапа синтеза: проектирование ПС, кодирование ПС, тестирование ПС (рисунок 3.7).

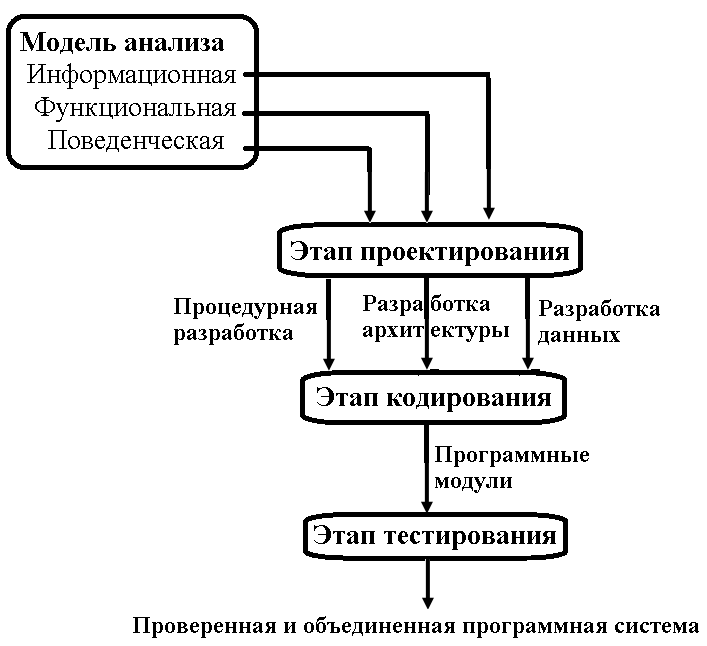


Рисунок 3.7 – Информационные потоки процесса синтеза ПС

Этап проектирования основывается на требованиях к ПС, представляется информационной, функциональной и поведенческой моделями анализа. Модели анализа представляют этапу проектирования исходные сведения для работы. Информационная модель описывает информацию, которую, по мнению заказчика, должна обрабатывать ПС. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы ее работы). На выходе этапа проектирования – разработка данных, разработка архитектуры и процедурная разработка ПС.

Разработка данных – это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации программной системы.

Разработка архитектуры выделяет основные структурные компоненты и фиксирует связи между ними.

Процедурная разработка описывает последовательность действий в структурных компонентах, таким образом, определяет их содержание.

Затем создаются тексты программных модулей, проводится тестирование для объединения и проверки ПС. На проектирование, кодирование и тестирование приходится более 75% стоимости конструирования ПС. Принятые здесь решения оказывают решающее воздействие на успех реализации ПС и легкость сопровождения в дальнейшем.

Решения, принимаемые в ходе проектирования, делают его стержневым этапом процесса синтеза. Проектирование – единственный путь, обеспечивающий правильную трансляцию требований заказчика в конечный программный продукт.

## Особенности этапа проектирования

Проектирование – итерационный процесс, при помощи которого требования к ПС транслируются в инженерные представления. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), последующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени – предварительное проектирование и детальное проектирование. Предварительное проектирование формирует абстракции архитектурного уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого – сформировать графический интерфейс пользователя (GUI). Схема информационных связей процесса проектирования приведена на рисунке 3.8.

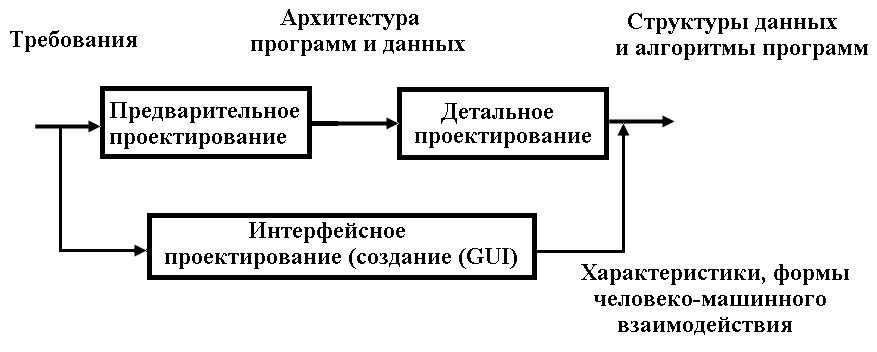


Рисунок 3.8 –Информационные связи процесса проектирования

Предварительное проектирование обеспечивает:

* + идентификацию подсистем;
  + определение основных принципов управления подсистемами, взаимодействия подсистем.

Предварительное проектирование включает три типа деятельности:

* + 1. *Структурирование системы.* Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем.
    2. *Моделирование управления.* Определяется модель связей управления между частями системы.
    3. *Декомпозиция подсистем на модули.* Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

## 

## Структурирование системы

Известны четыре модели системного структурирования:

* модель хранилища данных;
* модель клиент-сервер;
* трехуровневая модель;
* модель абстрактной машины.

В *модели хранилища данных* (рисунок 3.9) подсистемы разделяют данные, находящиеся в общей памяти. Как правило, данные образуют БД. Предусматривается система управления этой базой.

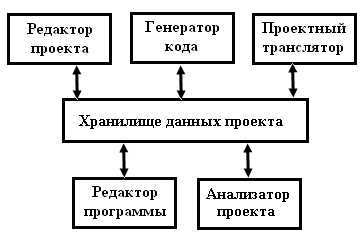


Рисунок 3.9 –Модель хранилища данных

*Модель клиент-сервер* используется для распределенных систем, где данные распределены по серверам (рисунок 3.10). Для передачи данных применяют сетевой протокол, например TCP/IP.

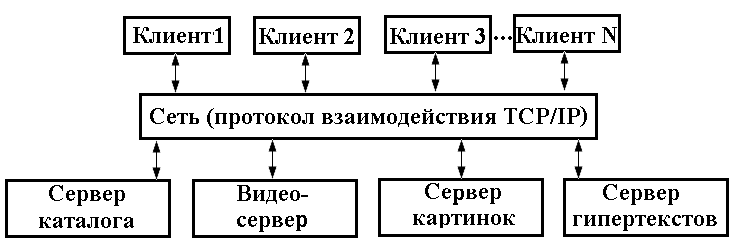


Рисунок 3.10 –Модель клиент-сервер

*Трехуровневая модель* является развитием модели клиент-сервер (рисунок 3.11).

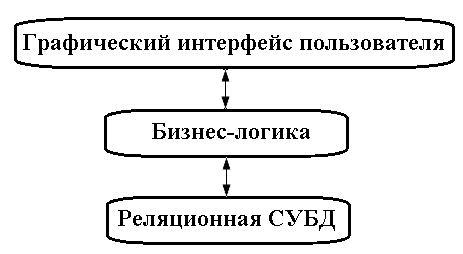


Рисунок 3.11 –Трехуровневая модель

Уровень графического интерфейса пользователя запускается на машине клиента. Бизнес-логику образуют модули, осуществляющие функциональные обязанности системы. Этот уровень запускается на сервере приложения. Реляционная СУБД хранит данные, требуемые уровню бизнес-логики. Этот уровень запускается на втором сервере – сервере базы данных.

Преимущества трехуровневой модели:

* упрощается такая модификация уровня, которая не влияет на другие уровни;
* отделение прикладных функций от функций управления БД упрощает оптимизацию всей системы.

*Модель абстрактной машины* отображает многослойную систему (рисунок 3.12).

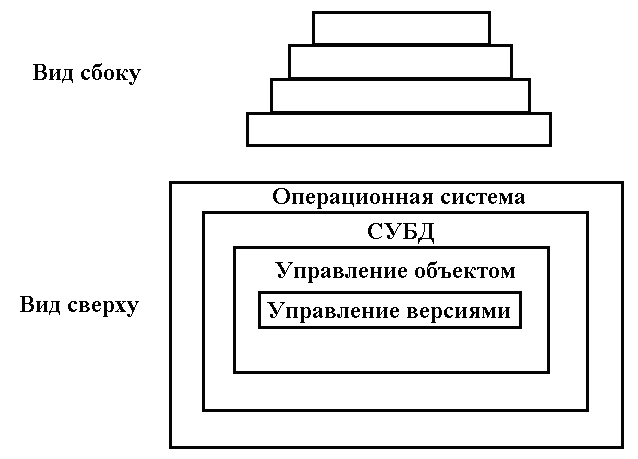


Рисунок 3.12 –Модель абстрактной машины

Каждый текущий слой реализуется с использованием средств, обеспечиваемых слоем-фундаментом.

## 

## Моделирование управления

Известны два типа моделей управления:

* модель централизованного управления;
* модель событийного управления.

В модели централизованного управления одна подсистема выделяется как системный контроллер. Ее обязанности – руководить работой других подсистем. Различают две разновидности моделей централизованного управления: *модель вызов-возврат* (рисунок 3.13) и *модель менеджера* (рисунок 3.14), которая используется в системах параллельной обработки.

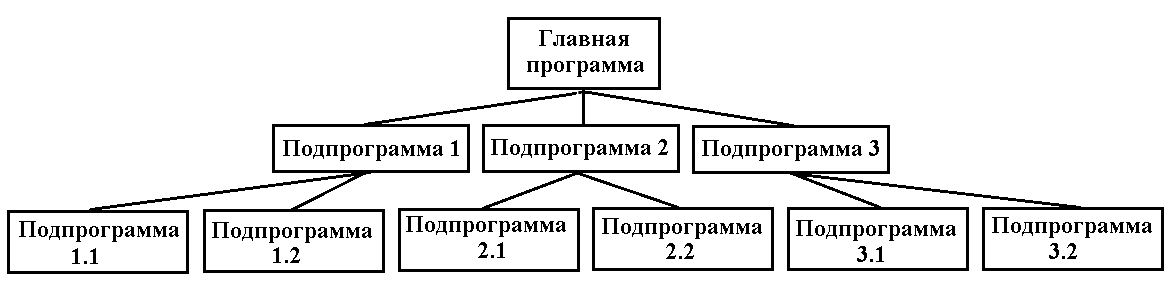


Рисунок 3.13 –Модель вызов-возврат

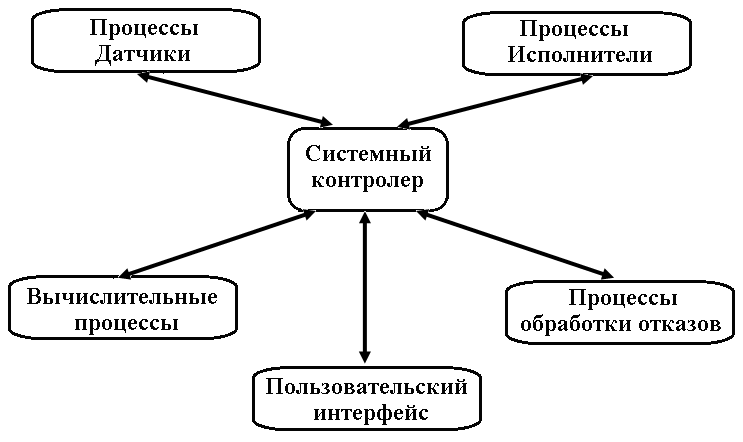


Рисунок 3.14 –Модель менеджера

В модели событийного управления системой управляют внешние события. Используются две разновидности модели событийного управления: широковещательная модель и модель, управляемая прерываниями.

В *широковещательной модели* (рисунок 3.15) каждая подсистема уведомляет обработчика о своем интересе к конкретным событиям. Когда событие происходит, обработчик пересылает его подсистеме, которая может обработать это событие. Функции управления в обработчик не встраиваются.



Рисунок 3.15 –Широковещательная модель

*В модели, управляемой прерываниями* (рисунок 3.16), все прерывания разбиты на группы – типы, которые образуют вектор прерываний. Для каждого типа прерывания есть свой обработчик. Каждый обработчик реагирует на свой тип прерывания и запускает свой процесс.

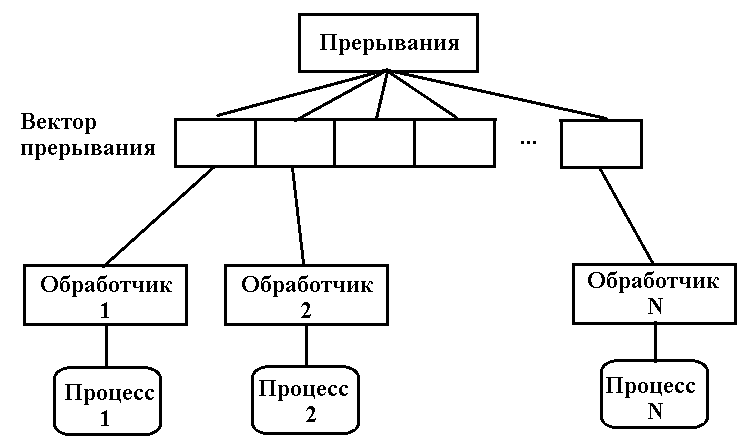


Рисунок 3.16 – Модель, управляемая прерываниями

## Декомпозиция подсистем на модули

Известны два типа моделей модульной декомпозиции:

* модель потока данных;
* модель объектов.

В основе модели потока данных лежит разбиение по функциям.

Модель объектов основана на слабо сцепленных сущностях, имеющих собственные наборы данных, состояния и наборы операций.

Выбор типа декомпозиции должен определяться сложностью разбиваемой подсистемы.

**3.3 Модульное программирование**

Модуль – фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

По определению Г. Майерса, модульность – свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы.

Модуль характеризуют:

* один вход и один выход. На входе программный модуль получает определенный набор исходных данных, выполняет их обработку и возвращает один набор выходных данных;
* функциональная завершенность. Модуль выполняет набор определенных операций для реализации каждой отдельной функции, достаточных для завершения начатой обработки данных;
* логическая независимость. Результат работы данного фрагмента программы не зависит от работы других модулей;
* слабые информационные связи с другими программными модулями. Обмен информацией между отдельными модулями должен быть минимален;
* размер и сложность программного элемента в разумных рамках.

Таким образом, модули содержат описание исходных данных, операции обработки данных и структуры взаимосвязи с другими модулями.

В настоящее время используют два способа декомпозиции разрабатываемого программного обеспечения, связанные с соответствующим подходом:

* процедурный (или структурный - по названию подхода);
* объектный.

Результатом процедурной декомпозиции является *иерархия подпрограмм* (процедур), в которой функции, связанные с принятием решения, реализуются подпрограммами верхних уровней, а непосредственно обработка – подпрограммами нижних уровней. Это согласуется с *принципом вертикального управления*, который был сформулирован вместе с другими рекомендациями структурного подхода к программированию. Он также ограничивает возможные варианты передачи управления, требуя, чтобы любая подпрограмма возвращала управление той подпрограмме, которая ее вызвала.

Результатом объектной декомпозиции является совокупность объектов, которые затем реализуют как переменные некоторых специально разрабатываемых типов (классов), представляющих собой совокупность полей данных и методов, работающих с этими полями.

Таким образом, при любом способе декомпозиции получают набор связанных с соответствующими данными подпрограмм, которые в процессе реализации организуют в модули.

Модулем называют автономно компилируемуюпрограммную единицу. Термин "модуль" традиционно используется в двух смыслах. Первоначально, когда размер программ был сравнительно невелик, и все подпрограммы компилировались отдельно, под модулем понималась подпрограмма, т. е. последовательность связанных фрагментов программы, обращение к которой выполняется по имени.Со временем, когда размер программ значительно вырос, и появилась возможность создавать библиотеки ресурсов: констант, переменных, описаний типов, классов и подпрограмм, термин "модуль" стал использоваться и в смысле *автономно компилируемый набор программных ресурсов.*

Данные модуль может получать и/или возвращать через общие области памяти или параметры.

**Основные характеристики программного модуля**

Не всякий программный модуль способствует упрощению программы. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Р. Хольт предложил следующие два общих критерия:

1. хороший модуль снаружи проще, чем внутри;
2. хороший модуль проще использовать, чем построить.

Г. Майерс предлагает для оценки приемлемости программного модуля использовать более конструктивные его характеристики:

* размер модуля,
* прочность модуля,
* сцепление с другими модулями,
* рутинность модуля (независимость от предыстории обращений к нему).

*Размер* модуля измеряется числом содержащихся в нем операторов или строк. Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

*Связность модуля*

Прочность (связность) модуля – это мера его внутренних связей. Связность модуля определяется как мера независимости его частей. Чем выше связность модуля, тем лучше результат проектирования. Для обозначения связности используется также понятие силы связности модуля. Типы связности модулей приведены в таблице 3.1.

Таблица 3.1 – Типы связности модулей

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вид связности | Сила связности | Наглядность  (понятность) | Возможность изменения | Сопровожда-емость |
| Функциональная | 10 | Хорошая | Хорошая | Хорошая |
| Последовательная | 9 | Хорошая | Хорошая | Хорошая |
| Информационная (коммуникативная) | 8 | Средняя | Средняя | Средняя |
| Процедурная | 5 | Средняя | Средняя | Плохая |
| Временная | 3 | Средняя | Средняя | Плохая |
| Логическая | 1 | Плохая | Плохая | Плохая |
| Случайная (по совпадению) | 0 | Плохая | Плохая | Плохая |

*Функционально прочный* модуль – это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции он может использовать и другие модули. Такие модули рекомендуются для использования. Модуль с функциональной связностью не может быть разбит на два других модуля, имеющих связность того же типа. Такой модуль реализуется последовательностью операций в виде единого цикла.

Функционально связный модуль содержит элементы, участвующие в выполнении одной и только одной проблемной задачи. Примеры функционально связных модулей:

* вычислять синус угла;
* проверять орфографию;
* читать запись файла;
* вычислять координаты цели;
* вычислять зарплату сотрудника;
* определять место пассажира;
* и др.

Каждый из этих модулей имеет единичное назначение. Когда клиент вызывает модуль, выполняется только одна работа без привлечения внешних обработчиков. Приложения, построенные из функционально связных модулей, легче сопровождать.

Модуль, имеющий *последовательную связность*, может быть разбит на последовательные части, выполняющие независимые функции, но совместно реализующие единую функцию. Если один и тот же модуль используется для оценки, а затем для обработки данных, то он имеет *последовательную связность*. Модуль с последовательной связностью реализуется как последовательность операций или последовательность циклов.

Пример.

Модуль *Прием и проверка записи*

прочитать запись из файла

проверить контрольные данные в записи

удалить контрольные поля в записи

вернуть обработанную запись

Конец *модуля.*

В этом модуле три элемента. Результаты первого элемента (прочитать запись из файла) используются как входные данные для второго элемента (проверить контрольные данные в записи) и т. д.

Сопровождать модули с информационной связностью не сложно, так же как и функционально связные модули. Но возможности повторного использования ниже, чем в случае функциональной связности, поскольку совместное применение действий модуля с информационной связностью не всегда приемлемо.

Если модуль составлен из независимых модулей, разделяющих структуру данных, он имеет *коммуникативную связность*. Общая структура данных является основой его организации как единого модуля. Если модуль спроектирован так, чтобы упростить работу со сложной структурой данных, изолировать эту структуру, он имеет коммуникативную связность. Такой модуль предназначен для выполнения нескольких различных и независимо используемых функций, таких как запоминание и поиск данных.

Пример модуля, имеющего коммуникативную связность.

Модуль *Отчет и средняя зарплата*

используется Таблица зарплаты служащих

сгенерировать Отчет по зарплате

вычислить параметр Средняя зарплата

вернуть Отчет по зарплате. Средняя зарплата

Конец *модуля*.

Здесь все элементы модуля работают со структурой Таблица зарплаты служащих.

Модули высшего уровня иерархической структуры должны иметь функциональную или последовательную связность. Для модулей обслуживания предпочтительнее коммуникативная связность.

*Процедурная связность* может возникнуть при расчленении длинной программы на части в соответствии с передачами управления, но без определения какого-либо функционального назначения. Процедурная связность может появиться при группировании альтернативных частей программы.

Процедурно связный модуль состоит из элементов, реализующих независимые действия, для которых задан порядок работы (порядок передачи управления). Зависимости по данным между элементами нет.

Пример.

Модуль *Вычисление средних значений*

используется Таблица-А. Таблица-В

вычислить среднее по Таблица-А

вычислить среднее по Таблица-В

вернуть среднееТабл-А, среднееТабл-В

Конец *модуля*.

Этот модуль вычисляет средние значения для двух полностью несвязанных таблиц Таблица-А и Таблица-В.

Модуль, содержащий части функционально не связанные, но необходимые в один и тот же момент обработки, имеет *временную связность* или связность *по классу*. Связность такого типа имеет место в тех случаях, когда все множество требуемых в момент входа в программу функций выполняется независимым модулем.

При связности по времени элементы-обработчики модуля привязаны к конкретному периоду времени (из жизни программной системы). Классическим примером временной связности является модуль инициализации:

Модуль *Инициализировать систему*

перемотать магнитную ленту 1

Счетчик магнитной ленты 1 := 0

перемотать магнитную ленту 2

Счетчик магнитной ленты 2 := 0

Таблица текущих записей : = пробел..пробел

Таблица количества записей := 0..0

Переключатель 1 : = выкл

Переключатель 2 := вкл

Конец *модуля*.

Элементы данного модуля почти не связаны друг с другом (за исключением того, что должны выполняться в определенное время). Они все – часть программы запуска системы. Зато элементы более тесно взаимодействуют с другими модулями, что приводит к сложным внешним связям.

Если в модуле объединены операторы только по признаку их функционального подобия (например, все они предназначены для проверки правильности данных или для управления операциями обмена с внешними носителями), а для его настройки применяется алгоритм переключения, такой модуль имеет *логическую связность*, поскольку его части ничем не связаны, а имеют лишь небольшое сходство между собой. Модуль, состоящий из разнообразных подпрограмм обработки ошибок, имеет логическую связность. Элементы логически связного модуля принадлежат к действиям одной категории, и из этой категории клиент выбирает выполняемое действие.

Пример.

Модуль *Пересылка сообщения*

переслать по электронной почте

переслать по факсу

послать в телеконференцию

переслать по ftp-протоколу

Конец *модуля*.

Если операторы модуля объединяются произвольным образом, например, когда необходимо указать их непосредственное размещение в области памяти, такой модуль имеет связность *по совпадению*. Связность по совпадению – самая слабая степень прочности модуля. Это такой модуль, между элементами которого нет осмысленных связей.

Пример.

Модуль *Разные функции* (какие-то параметры)

поздравить с Новым годом (...)

проверить исправность аппаратуры (...)

заполнить анкету героя (...)

измерить температуру (...)

запастись продуктами (...)

Конец *модуля*.

Такие модули не рекомендуются для использования.

При проектировании программных модулей лучше всего использовать функциональную, последовательную и информационную связность, три наиболее слабых типа связности (временная, логическая и по совпадению) возникают в результате неправильного планирования и проектирования ПО.

## Сцепление модулей

Сцепление модуля – это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей.

Сцепление модулей представляет собой меру относительной независимости модулей, которая определяет их читабельность и сохранность. Независимые модули могут быть модифицированы без переделки каких-либо других модулей. Слабое сцепление более желательно, т.к. это означает высокий уровень их независимости. Модули являются полностью *независимыми*, если каждый их них не содержит о другом никакой информации. Чем больше информации о других модулях используется в них, тем менее они независимы и тем теснее сцеплены. В таблице 3.2 приведены типы сцепления модулей.

Модули сцеплены *по данным (параметрическое сцепление)*, если они имеют общие единицы, которые передаются от одного к другому как параметры, представляющие собой простые элементы данных, то есть вызывающий модуль "знает" только имя вызываемого модуля, а также типы и значения некоторых его переменных.

Таблица 3.2 – Типы сцепления модулей

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип сцепления | Степень сцепления | Устойчивость  к ошибкам | Наглядность  (понятность) | Возможность  изменения | Вероятность  повторного  использования |
| По данным | 1 | Хорошая | Хорошая | Хорошая | Большая |
| По образцу | 3 | Средняя | Хорошая | Средняя | Средняя |
| По управлению | 4 | Средняя | Плохая | Плохая | Малая |
| По внешним ссылкам | 5 | Средняя | Плохая | Плохая | Малая |
| По общей области | 7 | Плохая | Плохая | Средняя | Малая |
| По содержимому (кодам) | 9 | Плохая | Плохая | Плохая | Малая |

Сцепление по данным(СЦ=1). Модуль А вызывает модуль В.

Все входные и выходные параметры вызываемого модуля – простые элементы данных (рисунок 3.17).

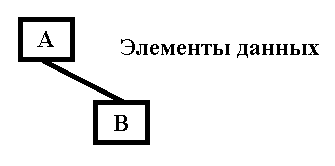


Рисунок 3.17 –Сцепление по данным

Например, функция Мах предполагает сцепление по данным через параметры скалярного типа:

**Function Max(a, b: integer): integer;**

**begin**

**if a>b then Max:=a**

**else Max: =b;**

**end;**

Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является *параметрическое сцепление* (сцепление по данным по Г. Майерсу) – это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Модули сцеплены *по образцу*, если параметры содержат структуры данных. Недостатком такого сцепления является то, что оба модуля должны знать о внутренней структуре данных друг друга. Если программист, сопровождающий программу, модифицирует структуру данных в одном из модулей, он вынужден изменить структуру данных и в другом модуле. Поэтому вероятность появления ошибок, возникающих при кодировании и сопровождении, повышается.

Сцепление по образцу(СЦ=3). В качестве параметров используются структуры данных (рисунок 3.18).

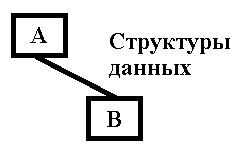


Рисунок 3.18 –Сцепление по образцу

Функция MaxEl предполагает сцепление по образцу (параметр **а** - открытый массив).

**Function MaxEl (a:array of integer): integer;**

**Var i:word;**

**begin**

**MaxEl: =a [0];**

**for i: =l to High (a) do**

**if a [i]>MaxEl then MaxEl: =a [i];**

end;

Модули имеют сцепление *по управлению*, если какой-либо из них управляет решениями внутри другого с помощью передачи флагов, переключателей или кодов, предназначенных для выполнения функций управления, то есть один из модулей знает о внутренних функциях другого. Установка флага, указывающего, какой именно способ доступа используется в операции обмена (последовательный или прямой), означает, что осуществляется сцепление по управлению.

Сцепление по управлению(СЦ=4). Модуль А явно управляет функционированием модуля В (с помощью флагов или переключателей), посылая ему управляющие данные (рисунок 3.19).

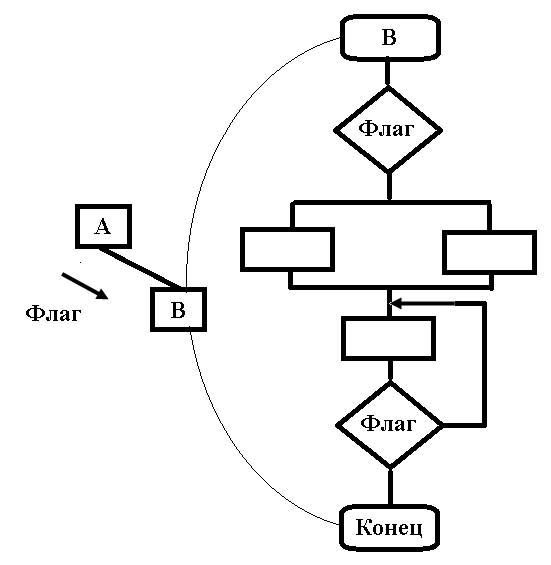


Рисунок 3.19 – Сцепление по управлению

Например, функция MinMax предполагает сцепление по управлению, так как значение параметра flag влияет на логику программы: если функция MinMax получает значение параметра flag, равное true, то возвращает максимальное значение из двух, а если false, то минимальное:

**Function MinMax (a, b: integer; flag:boolean): integer;**

**begin**

**if(a>b) and (flag) then MinMax: =a**

**else MinMax: =b;**

**end;**

Модуль сцеплен *по внешним ссылкам* (СЦ=5), если у него есть доступ к данным в другом модуле через внешнюю точку входа. Сцепление такого типа возникает, при использовании ПЛ/1 или Паскаля, когда внутренние процедуры оперируют с глобальными переменными (два модуля А и В ссылаются на один и тот же глобальный элемент данных).

Модули сцеплены *по общей области*, если они разделяют одну и ту же глобальную структуру данных (рисунок 3.20) (СЦ=7). В этом случае возможностей для появления ошибок при модификации структуры данных в одном из модулей намного больше.

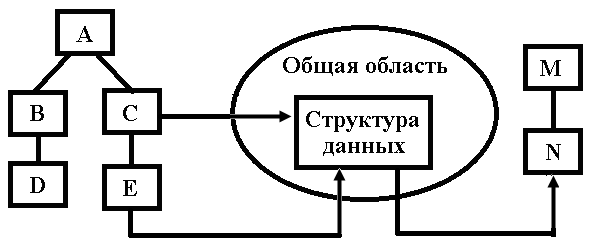


Рисунок 3.20 –Сцепление по общей области и содержанию

Этот тип сцепления считается недопустимым, поскольку:

* программы, использующие данный тип сцепления, очень сложны для понимания при сопровождении программного обеспечения;
* ошибка одного модуля, приводящая к изменению общих данных, может проявиться при выполнении другого модуля, что существенно усложняет локализацию ошибок;
* при ссылке к данным в общей области модули используют конкретные имена, что уменьшает гибкость разрабатываемого программного обеспечения.

Например, функция МахА, использующая глобальный массив А, сцеплена с основной программой по общей области:

**Function MaxA: integer;**

**Var i:word;**

**begin**

**МахА: =a[Low(a)];**

**for i: = Low (a) + l to High(a) do**

**if a [i]>MaxA then MaxA: = a [i];**

**end;**

Подпрограммы с "памятью", действия которых зависят от истории вызовов, используют сцепление по общей области, что делает их работу в общем случае непредсказуемой. Именно этот вариант используют статические переменные С и C++.

*Сцепление по содержанию (кодам)*(СЦ=9). Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). На рисунке 3.20 видим, что модули В и D сцеплены по содержанию, а модули С, Е и N сцеплены по общей области.

При таком сцепленииодин модуль содержит обращения к внутренним компонентам другого (передает управление внутрь, читает и/или изменяет внутренние данные или сами коды), что полностью противоречит блочно-иерархическому подходу. Отдельный модуль в этом случае уже не является блоком ("черным ящиком"), его содержимое должно учитываться в процессе разработки другого модуля. Современные универсальные языки процедурного программирования, например Pascal, данного типа сцепления в явном виде не поддерживают, но для языков низкого уровня, например Ассемблера, такой вид сцепления остается возможным.

*Рутинность* модуля – это его независимость от предыстории обращений к нему. Модуль называют *рутинным*, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль называют *зависящим от предыстории*, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему. Г. Майерс не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах ошибок, которые сложно найти. Но такая рекомендация иногда является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому приемлема следующая рекомендация:

1. всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;
2. зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
3. в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

*Порядок разработки программного модуля*

При разработке программного модуля целесообразно придерживаться следующего порядка:

1. изучение и проверка спецификации модуля, выбор языка программирования;
2. выбор алгоритма и структуры данных;
3. программирование (кодирование) модуля;
4. шлифовка текста модуля;
5. проверка модуля;
6. компиляция модуля.

**Библиотеки ресурсов**

Различают библиотеки ресурсов двух типов: библиотеки подпрограмм и библиотеки классов.

Библиотеки подпрограмм реализуют функции, близкие по назначению, например, библиотека графического вывода информации. Связность подпрограмм между собой в такой библиотеке – логическая, а связность самих подпрограмм – функциональная, так как каждая из них обычно реализует одну функцию.

Библиотеки классов реализуют близкие по назначению классы. Связность элементов класса – информационная, связность классов между собой может быть функциональной для родственных или ассоциированных классов и логической для остальных.

В качестве средства улучшения технологических характеристик библиотек ресурсов широко используют разделение тела модуля на интерфейсную часть и область реализации.

*Интерфейсная часть* содержит совокупность объявлений ресурсов (заголовков подпрограмм, имен переменных, типов, классов и т. п.), которые данная библиотека предоставляет другим модулям. Ресурсы, объявление которых в интерфейсной части отсутствует, извне не доступны. Область *реализации* содержит тела подпрограмм и, возможно, внутренние ресурсы (подпрограммы, переменные, типы), используемые этими подпрограммами. При такой организации любые изменения реализации библиотеки, не затрагивающие ее интерфейс, не требуют пересмотра модулей, связанных с библиотекой, что улучшает технологические характеристики модулей-библиотек. Кроме того, подобные библиотеки, как правило, хорошо отлажены и продуманы, так как часто используются разными программами.

**Модульная структура программных продуктов**

Модульная структура ПО представляет собой древовидную структуру, в узлах которой размещаются программные модули, а направленные дуги показывают статическую подчиненность модулей. Если в тексте модуля имеется ссылка на другой модуль, то их на структурной схеме соединяет дуга, которая исходит из первого и входит во второй модуль. Другими словами, каждый модуль может обращаться к подчиненным ему модулям. При этом модульная структура программной системы должна включать в себя совокупность спецификаций модулей, образующих эту систему.

*Спецификация* программного модуля содержит:

* синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу),
* функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов).

Функциональная спецификация модуля строится так же, как и функциональная спецификация ПС.

Функция верхнего уровня обеспечивается главным модулем, он управляет выполнением нижестоящих функций, которым соответствуют подчиненные модули. При определении набора модулей, реализующих функции конкретного алгоритма, необходимо учитывать следующее:

1) модуль вызывается на выполнение вышестоящим по иерархии модулем и, закончив работу, возвращает ему управление;

2) принятие основных решений в алгоритме выносится на максимально высокий по иерархии уровень;

3) если в разных местах алгоритма используется одна и та же функция, то она оформляется в отдельный модуль, который будет вызываться по мере необходимости.

Состав, назначение и характер использования программных модулей в значительной степени определяются инструментальными средствами.

В процессе разработки ПС ее модульная структура может формироваться по-разному и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры ПО. Обычно в литературе обсуждаются два метода это метод восходящей разработки и метод нисходящей разработки.

Метод *восходящей разработки* заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (восходящем) порядке, в каком велось их программирование. Такой порядок разработки программы на первый взгляд кажется вполне естественным, каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Однако, современная технология не рекомендует такой порядок разработки программы. Во-первых, для программирования какого-либо модуля не требуется наличия текстов используемых им модулей – для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объеме, позволяющем построить правильное обращение к нему), а для его тестирования используемые модули заменять их имитаторами (заглушками, драйверами). Во-вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальным для ее модулей соображениям (принципам реализации, предположениям, структурам данных и т.п.), что определяет ее концептуальную целостность и формируется в процессе ее разработки. При восходящей разработке эта глобальная информация для модулей нижних уровней еще не ясна в полном объеме, поэтому очень часто их приходится перепрограммировать, когда при программировании других модулей производится существенное уточнение этой глобальной информации (например, изменяется глобальная структура данных). В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу (модуль), которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему. Это приводит к большому объему "отладочного" программирования и в то же время не дает никакой гарантии, что тестирование модулей производилось именно в тех условиях, в которых они будут выполняться в рабочей программе.

Подход имеет следующие недостатки:

* увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;
* наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;
* позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т. д.

Исторически восходящий подход появился раньше, что связано с особенностью мышления программистов, которые в процессе обучения привыкают при написании небольших программ сначала детализировать компоненты нижних уровней (подпрограммы, классы). Это позволяет им лучше осознавать процессы верхних уровней. При промышленном изготовлении программного обеспечения восходящий подход в настоящее время практически не используют.

Метод *нисходящей разработки* заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно и исключается перепрограммирование модулей. Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при ее применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, выдумывая абстрактные операции, которые позже нужно будет реализовать с помощью выделенных в программе модулей.

При использовании нисходящего подхода применяют иерархический, операционный, комбинированный методы и метод целенаправленной конструктивной реализации. Эти методы определяют последовательность проектирования и реализации компонентов.

*Иерархический метод* предполагает выполнение разработки строго по уровням. Исключения допускаются при наличии зависимости по данным, т. е. если обнаруживается, что некоторый модуль использует результаты другого, то его рекомендуется программировать после этого модуля. Основной проблемой данного метода является большое количество достаточно сложных заглушек. Кроме того, при использовании данного метода основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.

*Операционный метод* связывает последовательность выполнения при запуске программы. Применение метода усложняется тем, что порядок выполнения модулей может зависеть от данных. Кроме того, модули вывода результатов, несмотря на то, что они вызываются последними, должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании. С точки зрения распределения человеческих ресурсов сложным является начало работ, пока не закончены все модули, находящиеся на так называемом *критическом* пути.

*Комбинированный метод* учитывает следующие факторы, влияющие на последовательность разработки:

* достижимость модуля – наличие всех модулей в цепочке вызова данного модуля;
* зависимость по данным – модули, формирующие некоторые данные, должны создаваться раньше обрабатывающих;
* обеспечение возможности выдачи результатов – модули вывода результатов должны создаваться раньше обрабатывающих;
* готовность вспомогательных модулей должна быть раньше обрабатывающих;
* наличие необходимых ресурсов.

Метод *целенаправленной конструктивной реализации* заключается в следующем. Сначала реализуются только те модули, которые необходимы для самого простейшего варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, сигнализацию о выходе за пределы этого частного случая. Затем к этой программе добавляются реализации некоторых других модулей (в частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемого программного обеспечения. Таким образом, обход дерева ПС производится с целью кратчайшим путем реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой ПС.

Кроме того, при прочих равных условиях сложные модули должны разрабатываться прежде простых, так как при их проектировании могут выявиться неточности в спецификациях, и чем раньше это произойдет, тем лучше.

Нисходящий подход допускает нарушение нисходящей последовательности разработки компонентов в специально оговоренных случаях. Так, если некоторый компонент нижнего уровня используется многими компонентами более высоких уровней, то его рекомендуют проектировать и разрабатывать раньше, чем вызывающие его компоненты. И, наконец, в первую очередь проектируют и реализуют компоненты, обеспечивающие обработку правильных данных, оставляя компоненты обработки неправильных данных на более поздний период.

Нисходящий подход обычно используют и при объектно-ориентированном программировании. В соответствии с рекомендациями подхода вначале проектируют и реализуют пользовательский интерфейс программного обеспечения, затем разрабатывают классы некоторых базовых объектов предметной области, а уже потом, используя эти объекты, проектируют и реализуют остальные компоненты.

Нисходящий подход обеспечивает:

* максимально полное определение спецификаций проектируемого компонента и согласованность компонентов между собой;
* раннее определение интерфейса пользователя, демонстрация которого заказчику позволяет уточнить требования к создаваемому программному обеспечению;
* возможность нисходящего тестирования и комплексной отладки.

## 

## Характеристики иерархической структуры программной системы

Иерархическая структура программной системы – основной результат предварительного проектирования. Она определяет состав модулей ПС и управляющие отношения между модулями. Иерархическая структура не отражает процедурные особенности программной системы, то есть последовательность операций, их повторение, ветвления и т. д. Основные характеристики иерархической структуры представлены на рисунке 3.21.

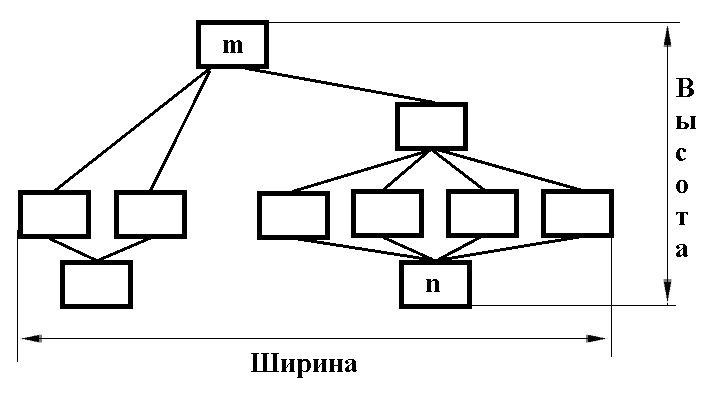


Рисунок 3.21 – Иерархическая структура программной системы

Первичными характеристиками являются количество вершин (модулей) и количество ребер (связей между модулями). К ним добавляются две глобальные характеристики – высота и ширина:

* высота– количество уровней управления (в примере высота 4);
* ширина– максимальное количество модулей, размещенных на уровнях управления (в примере ширина 6).

Локальными характеристиками модулей структуры являются коэффициент объединения по входу и коэффициент разветвления по выходу.

Коэффициент объединения по входу Fan\_in(*i*) – это количество модулей, которые прямо управляют *i*-м модулем (в примере на рисунке 3.21 для модуля n: Fan\_in(*n*)=4).

Коэффициент разветвления по выходу Fan\_out(*i*) – это количество модулей, которыми прямо управляет *i*-й модуль (для модуля m: Fan\_out(*m*)=3).

**3.4 Структурное программирование**

Сущность структурного подхода к разработке ИС заключается в ее декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и так далее. Процесс разбиения продолжается вплоть до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны.

Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов:

* принцип "разделяй и властвуй" – принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;
* принцип иерархического упорядочивания – принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.
* принцип абстрагирования заключается в выделении существенных аспектов системы и отвлечения от несущественных;
* принцип формализации заключается в необходимости строгого методического подхода к решению проблемы;
* принцип непротиворечивости заключается в обоснованности и согласованности элементов;
* принцип структурирования данных заключается в том, что данные должны быть структурированы и иерархически организованы.

Э. Дейкстра предложил строить программу как композицию из нескольких типов управляющих конструкций (структур), которые позволяют повысить понимаемость логики работы программы. Программирование с использованием только таких конструкций назвали *структурным*.

Для изображения передач управления в модулях используются структурные схемы программ (рисунок 3.22)

Основными конструкциями структурного программирования являются: следование (выполнение операторов последовательно), разветвление (в зависимости от выполнения некоторого условия выполняется та или иная последовательность операторов) и повторение (многократное выполнение одинаковой последовательности операторов) (см. рисунок 3.22). Существенно, что каждая из этих конструкций имеет по управлению только один вход и один выход. Весьма важно также, что эти конструкции являются уже математическими объектами (что, по существу, и объясняет причину успеха структурного программирования).

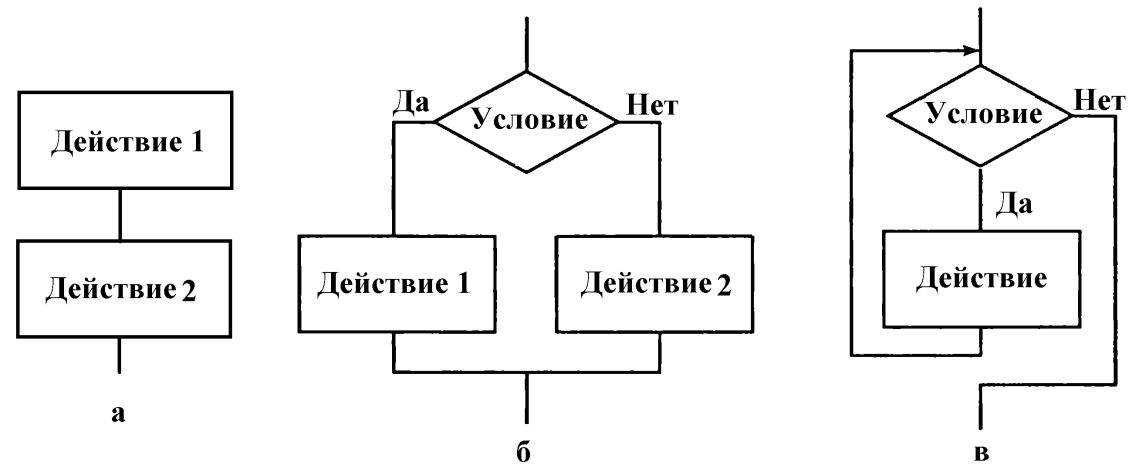


Рисунок 3.22 – Базовые алгоритмические структуры: а – следование; б – ветвление; в – цикл-пока

Спецификации процессов могут быть представлены в виде псевдокодов, блок-схем алгоритмов, Flow-форм, диаграмм Насси – Шнейдермана или просто краткого текстового описания.

Наибольшую популярность получили схемы алгоритмов.

Для изображения схем алгоритмов разработан ГОСТ 19.701–90 (рисунок 3.23).

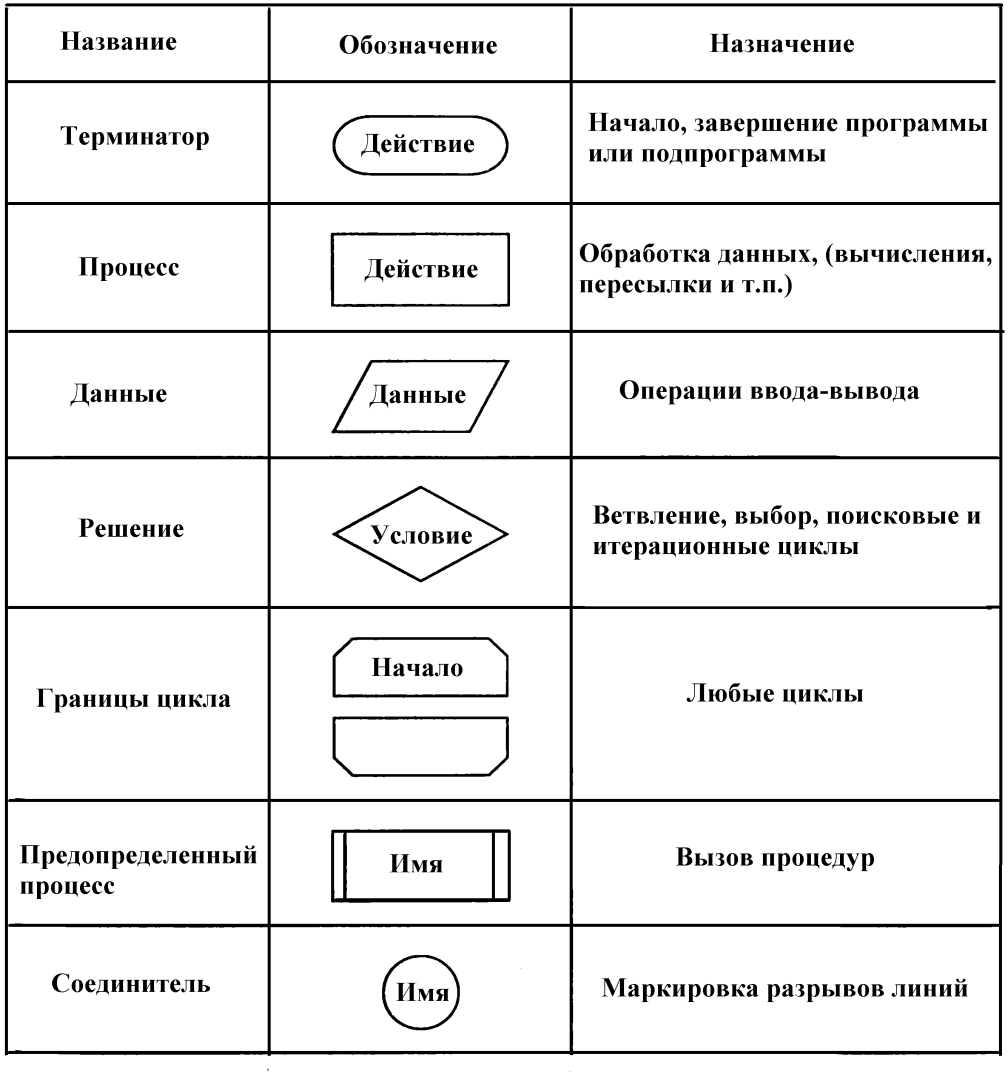


Рисунок 3.23 – Обозначение элементов схем алгоритмов

**Анализ требований и определение спецификаций при структурном подходе**

На этом этапе необходимо построить модели ПО во взаимодействии с окружающей средой. Поскольку разные модели описывают проектируемое программное обеспечение с разных сторон, рекомендуется использовать сразу несколько моделей и сопровождать их описаниями. Структурный подход к проектированию программных продуктов предполагает разработку следующих моделей:

* диаграмм потоков данных (DFD – Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе;
* диаграмм "сущность–связь" (ERD Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы;
* диаграмм переходов состояний (STD – State Transition Diagrams), характеризующих поведение системы во времени;
* функциональных диаграмм (методология SADT – Structured Analysis and Design Technique) методологией структурного анализа и проектирования.

**Диаграммы потоков данных (DFD)**

Такая диаграмма состоит из трех типов узлов: узлов обработки данных, узлов хранения данных и внешних узлов, представляющих внешние по отношению к используемой диаграмме источники или потребители данных. Дуги в диаграмме соответствуют потокам данных, передаваемых от узла к узлу. Они помечены именами соответствующих данных. Описание процесса, функции или системы обработки данных, соответствующих узлу диаграммы, может быть представлено диаграммой следующего уровня детализации, если процесс достаточно сложен.

Для изображения диаграмм потоков данных традиционно используют два вида нотаций: нотации Йордана и Гейна – Сарсона (рисунок 3.24).



Рисунок 3.24 – Обозначения элементов диаграмм потоков данных

Первым шагом при построении иерархии диаграмм потоков данных является построение контекстных диаграмм, показывающих, как система будет взаимодействовать с пользователями и другими внешними системами. При проектировании простых систем достаточно одной контекстной диаграммы, имеющей звездную топологию, в центре которой располагается основной процесс, соединенный с источниками и приемниками информации.

Для сложных систем строится иерархия контекстных диаграмм, которая определяет взаимодействие основных функциональных подсистем проектируемой системы как между собой, так и с внешними входными и выходными потоками данных и внешними объектами. При этом контекстная диаграмма верхнего уровня содержит набор подсистем, соединенных потоками данных. Контекстные диаграммы следующего уровня детализируют содержимое и структуру подсистем.

Процесс построения модели разбивается на несколько этапов.

1. Выделение множества требований в основные функциональные группы – процессы.

2. Выявление внешних объектов, связанных с разрабатываемой системой.

3. Идентификация основных потоков информации, циркулирующей между системой и внешними объектами.

4. Предварительная разработка контекстной диаграммы.

5. Проверка предварительной контекстной диаграммы и внесение в нее изменений.

6. Построение контекстной диаграммы путем объединения всех процессов предварительной диаграммы в один процесс, а также группирования потоков.

7. Проверка основных требований контекстной диаграммы.

8. Декомпозиция каждого процесса текущей DFD с помощью детализирующей диаграммы или спецификации процесса.

9. Проверка основных требований по DFD соответствующего уровня.

10. Добавление определений новых потоков в словарь данных при каждом их появлении на диаграммах.

11. Проверка полноты и наглядности модели после построения каждых двух-трех уровней.

**Диаграммы "сущность**-**связь"**

Базовыми понятиями ER-модели данных (ER – Entity- Relationship) являются сущность, атрибут и связь.

Первый вариант модели "сущность-связь" был предложен в 1976 г. Питером Пин-Шэн Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей (нотация Р. Мартина, нотация IDEF1X, нотация Р. Баркера и др.). Кроме того, различные программные средства, реализующие одну и ту же нотацию, могут отличаться своими возможностями. Все варианты диаграмм "сущность-связь" исходят из одной идеи – рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов) и взаимосвязей между сущностями.

Нотация Р. Баркера является наиболее распространенной.

Сущность – это класс однотипных объектов, информация о которых должна быть учтена в модели. Сущность имеет наименование, выраженное существительным в единственном числе, и обозначается в виде прямоугольника с наименованием (рисунок 3.25, а). Примерами сущностей могут быть такие классы объектов, как "Студент", "Сотрудник", "Товар".

*Экземпляр сущности* – это конкретный представитель данной сущности. Например, конкретный представитель сущности "Студент" – "Максимов". Причем сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности, для того чтобы различать экземпляры.

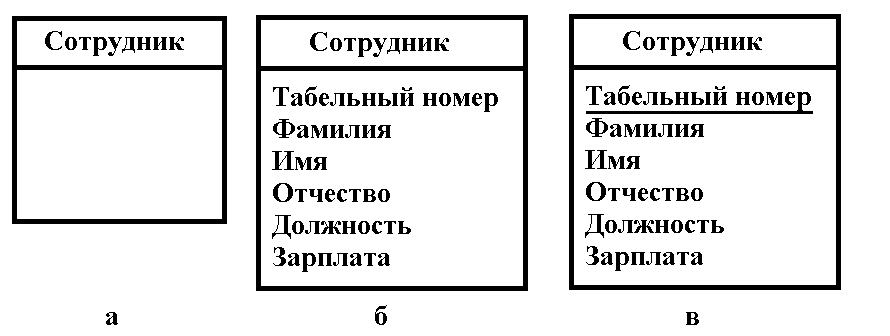


Рисунок 3.25 – Обозначения сущности в нотации Баркера: а – без атрибутов; б – с указанием атрибутов; в – с ключевым атрибутом

*Атрибут сущности* – это именованная характеристика, являющаяся некоторым свойством сущности. Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с описательными оборотами или прилагательными). Примерами атрибутов сущности "Студент" могут быть такие атрибуты, как "Номер зачетной книжки", "Фамилия", "Имя", "Пол", "Возраст", "Средний балл" и т. п. Атрибуты изображаются в прямоугольнике, обозначающем сущность (рисунок 3.25, б).

*Ключ сущности* – это не избыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра сущности. При удалении любого атрибута из ключа нарушается его уникальность. Ключей у сущности может быть несколько. На диаграмме ключевые атрибуты отображаются подчеркиванием (рисунок 3.25, в).

*Связь* – это отношение одной сущности к другой или к самой себе. Возможно по одной сущности находить другие, связанные с ней. Например, связи между сущностями могут выражаться следующими фразами – "СОТРУДНИК может иметь несколько ДЕТЕЙ", "СОТРУДНИК обязан числиться точно в одном ОТДЕЛЕ". Графически связь изображается линией, соединяющей две сущности (рисунок 3.26).

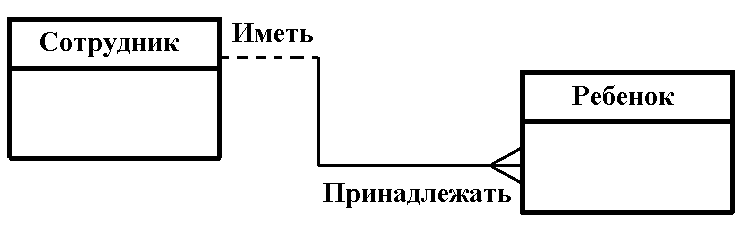


Рисунок 3.26 – Пример связи между сущностями

Каждая связь имеет одно или два наименования. Наименование обычно выражается неопределенной формой глагола: "Продавать", "Быть проданным" и т. п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности.

Связь может иметь один из следующих типов – рисунок 3.27.

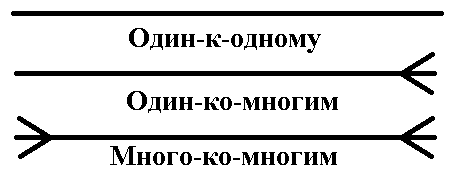


Рисунок 3.27 – Типы связей

Связь типа "один-к-одному" означает, что один экземпляр первой сущности связан точно с одним экземпляром второй сущности. Такая связь чаще всего свидетельствует о том, что мы неправильно разделили одну сущность на две.

Связь типа "один-ко-многим" означает, что один экземпляр первой сущности связан с несколькими экземплярами второй сущности. Это наиболее часто используемый тип связи. Пример такой связи приведен на рисунке 3.26.

Связь типа "много-ко-многим" означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и наоборот. Тип связи "много-ко-многим" является временным типом связи, допустимым на ранних этапах разработки модели. В дальнейшем такую связь необходимо заменить двумя связями типа "один-ко-многим" путем создания промежуточной сущности.

Каждая связь может иметь одну из двух модальностей связи (рисунок 3.28).

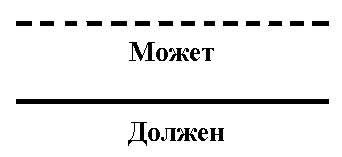


Рисунок 3.28 – Модальности связей

Связь может иметь разную модальность с разных концов, как показано на рисунке 3.26. Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на рисунке 3.26 читается так:

* слева направо: "Сотрудник может иметь несколько детей";
* справа налево: "Ребенок должен принадлежать точно одному сотруднику".

**Диаграммы переходов состояний (SDT- State Transition Diagrams)**

SDT демонстрирует поведение разрабатываемой программной системы при получении управляющих воздействий (извне).

В диаграммах такого вида узлы соответствуют состояниям динамической системы, а дуги – переходу системы из одного состояния в другое. Узел, из которого выходит дуга, является начальным состоянием, узел, в который дуга входит, – следующим. Дуга помечается именем входного сигнала или события, вызывающего переход, а также сигналом или действием, сопровождающим переход. Условные обозначения, используемые при построении диаграмм переходов состояний, показаны на рисунке 3.29.

На рисунке 3.30 представлена диаграмма переходов состояний программы, активно не взаимодействующей с окружающей средой, которая имеет примитивный интерфейс, производит некоторые вычисления и выводит простой результат.

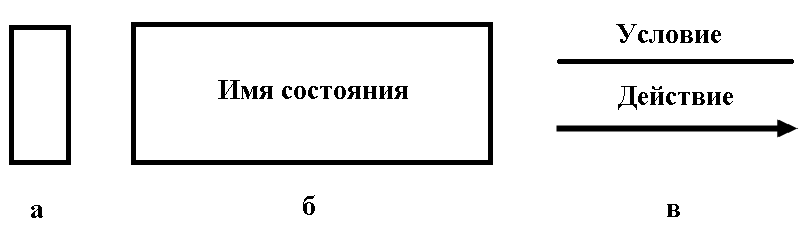


Рисунок 3.29 – Условные обозначения диаграмм переходов состояний: a – терминальное состояние; б – промежуточное состояние; в – переход

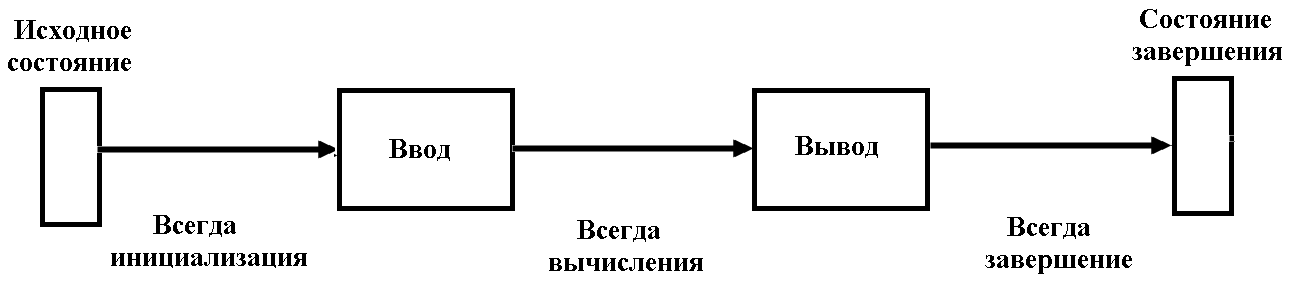


Рисунок 3.30 – Диаграмма переходов состояний программного обеспечения, активно не взаимодействующего с окружающей средой

На рисунке 3.31 представлена диаграмма переходов торгового автомата, активно взаимодействующего с покупателем.



Рисунок 3.31 – Диаграмма переходов состояний торгового автомата

**Функциональные диаграммы SADT (Structured Analysis and Design Technique)**

Функциональными называют диаграммы, в первую очередь отражающие взаимосвязи функций разрабатываемого программного обеспечения.

Они создаются на ранних этапах проектирования систем, для того чтобы помочь проектировщику выявить основные функции и составные части проектируемой системы и, по возможности, обнаружить и устранить существенные ошибки. Современные методы структурного анализа и проектирования предоставляют разработчику определенные синтаксические и графические средства проектирования функциональных диаграмм информационных систем.

В качестве примера рассмотрим методологию SADT, предложенную Дугласом Россом. На ее основе разработана известная методология IDEF0 (Icam DEFinition). Методология SADT представляет собой набор методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области.

Функциональная модель SADT отображает функциональную структуру объекта, т. е. производимые им действия и связи между этими действиями. Основные элементы этой методологии основываются на следующих концепциях:

* графическое представление блочного моделирования. На SADT-диаграмме функции представляются в виде блока, а интерфейсы входа-выхода – в виде дуг, соответственно входящих в блок и выходящих из него. Интерфейсные дуги отображают взаимодействие функций друг с другом;
* строгость и точность отображения.

Правила SADT включают:

* уникальность меток и наименований;
* ограничение количества блоков на каждом уровне декомпозиции;
* синтаксические правила для графики;
* связность диаграмм;
* отделение организации от функции;
* разделение входов и управлений.

Методология SADT может использоваться для моделирования и разработки широкого круга систем, удовлетворяющих определенным требованиям и реализующих требуемые функции. В уже разработанных системах методология SADT может быть использована для анализа выполняемых ими функций, а также для указания механизмов, посредством которых они осуществляются.

Диаграммы – главные компоненты модели, все функции программной системы и интерфейсы на них представлены как блоки и дуги. Место соединения дуги с блоком определяет тип интерфейса. Дуга, обозначающая управление, входит в блок сверху, в то время как информация, которая подвергается обработке, представляется дугой с левой стороны блока, а результаты обработки – дугами с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется в виде дуги, входящей в блок снизу (рисунок 3.32).

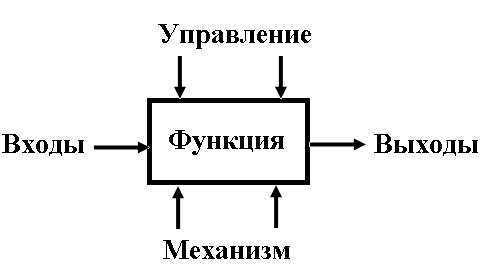


Рисунок 3.32 – Функциональный блок и интерфейсные дуги

Блоки на диаграмме размещают по "ступенчатой" схеме в соответствии с последовательностью их работы или доминированием, которое понимается как влияние, оказываемое одним блоком на другие. В функциональных диаграммах SADT различают пять типов влияний блоков друг на друга:

* вход-выход блока подается на вход блока с меньшим доминированием, т. е. следующего (рисунок 3.33, а);
* управление. Выход блока используется как управление для блока с меньшим доминированием (рисунок 3.33, б),
* обратная связь по входу. Выход блока подается на вход блока с большим доминированием (рисунок 3.33, в);
* обратная связь по управлению. Выход блока используется как управляющая информация для блока с большим доминированием (рисунок 3.33, г);
* выход-исполнитель. Выход блока используется как механизм для другого блока (рисунок 3.33, д).

Одной из наиболее важных особенностей методологии SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

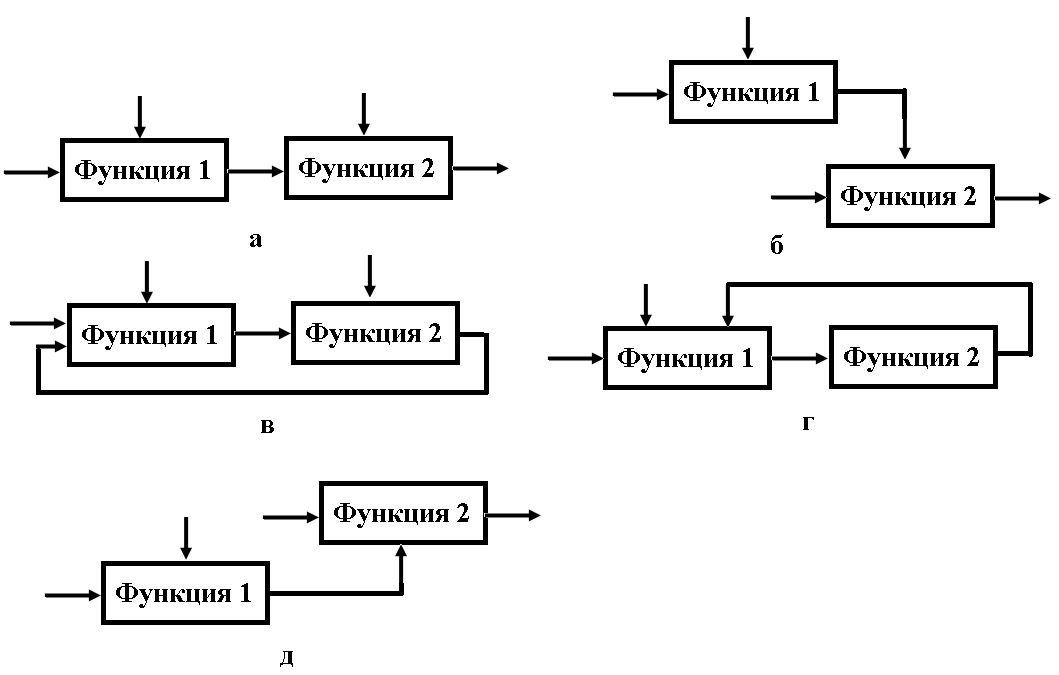


Рисунок 3.33 – Типы влияний блоков: а – вход; б – управление; в – обратная связь по входу; г – обратная связь по управлению; д – выход-исполнитель

При проектировании сложного программного обеспечения прежде всего необходимо определить структурные компоненты и связи между ними. Полученная в результате структура ПО должна быть представлена в виде структурной или функциональной схем и спецификаций ее компонентов.

**Функциональная схема**

Функциональная схема (ГОСТ 19.701–90) – это схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств. Для изображения функциональных схем используют специальные обозначения, установленные стандартом (рисунок 3.34).

Функциональные схемы более информативны, чем структурные. На рисунке 3.35 приведена укрупненная функциональная схема программной системы "Биржа труда".

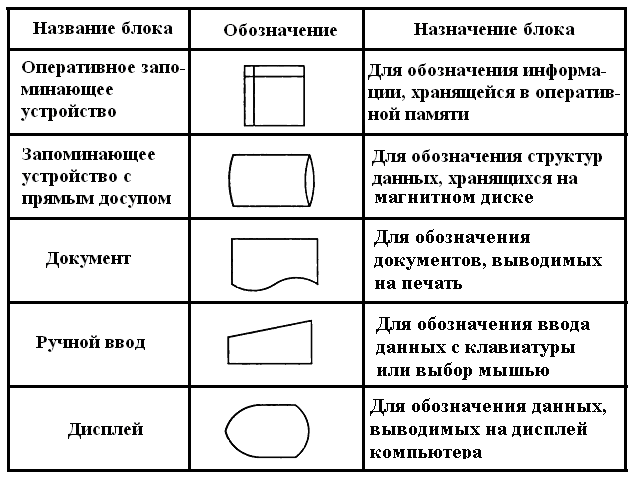


Рисунок 3.34 – Обозначения элементов функциональных схем

Функциональные схемы состоят их нескольких прямоугольных блоков, содержащих название программ (модулей в зависимости от степени детализации). Эти блоки соединяются входящими в них стрелками с источниками и исходящими из них стрелками – с приемниками данных. Источники и приемники изображаются в виде блоков, как принято согласно ЕСПД (Единая система программной документации). Основное внимание в схемах этого типа уделяется функциям, которые реализует ПС и используемым наборам данных.

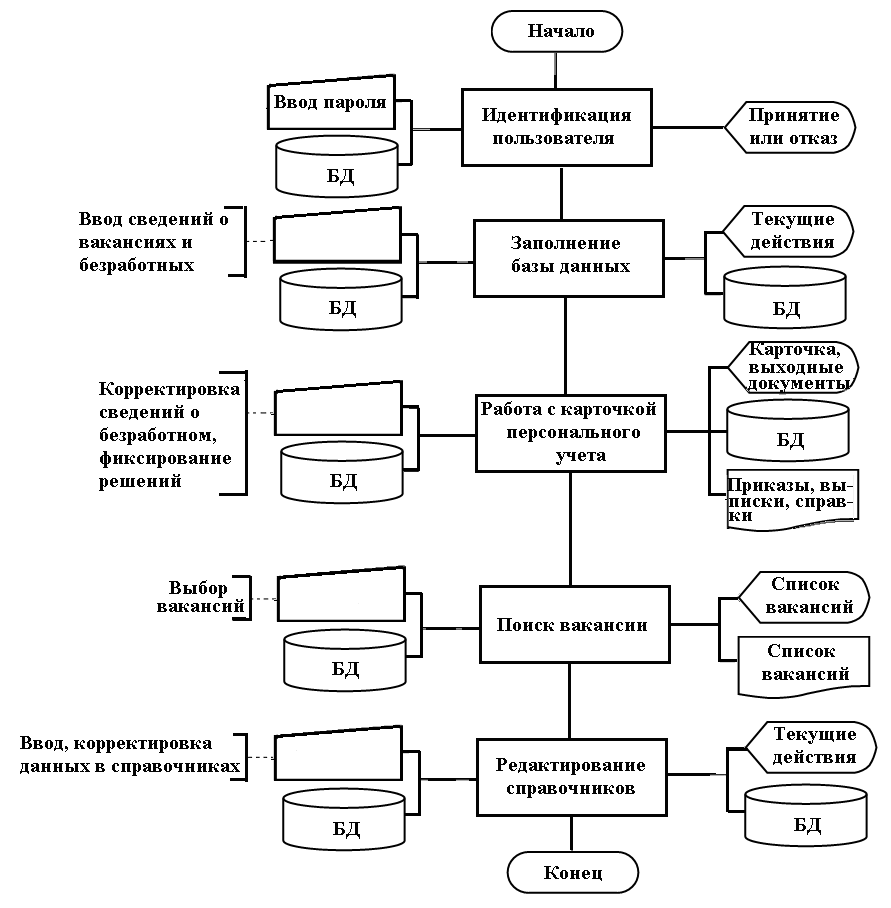


Рисунок 3.35 – Функциональная схема программной системы "Биржа труда"

**Схемы Джексона**

Техника диаграмм Джексона основана на методе структурного программирования, который выявляет соответствие между структурой потоков данных и структурой программы. Основное внимание в методе сконцентрировано на соответствии входных и выходных потоков данных. Структуры на диаграммах Джексона строятся из четырех основных компонентов, представленных на рисунке 3.36:

* операция – блок кодов, имеющий один вход и один выход (рис. 3.36, а);
* следование – последовательное выполнение операций слева направо (рис. 3.36, б);
* выбор – выполнение одной из операций в зависимости от выполнения условия (рис. 3.36, в);
* итерация – многократное выполнение блока (обозначается \* рис. 3.36, г).

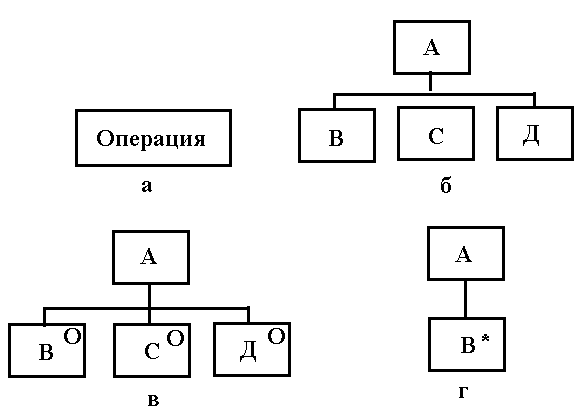


Рисунок 3.36 – Элементы структурных диаграмм Джексона

**Метод пошаговой детализации или уточнения**

Сущность этого метода заключается в разбиении процесса разработки текста модуля на ряд шагов. На первом шаге описывается общая схема работы модуля в обозримой линейной текстовой форме (т.е. с использованием очень крупных понятий), причем это описание не является полностью формализованным и ориентировано на восприятие его человеком. На каждом следующем шаге производится уточнение и детализация одного из понятий (оно называется *уточняемым*), в каком либо описании, разработанном на одном из предыдущих шагов. Запись осуществляется на псевдокоде – операторы выбранного языка программирования и текст на языке разработчика (в нашем случае на русском). Этот процесс завершается, когда все уточняемые понятия будут уточнены. Головным описанием на псевдокоде можно считать внешнее оформление модуля на базовом языке программирования, которое должно содержать:

1. начало модуля на базовом языке, т.е. первое предложение или заголовок (спецификацию) этого модуля ;
2. раздел (совокупность) описаний на базовом языке, причем вместо описаний процедур и функций – только их внешнее оформление;
3. неформальное обозначение последовательности операторов тела модуля как одного обобщенного оператора, а также неформальное обозначение тела каждого описания процедуры или функции как одного обобщенного оператора;
4. последнее предложение (конец) модуля на базовом языке.

Метод пошаговой детализации реализует нисходящий подход к программированию.

Пример.

Требуется определить наименьшее значение в некотором наборе данных и вывести эти данные, поделенные на наименьшее значение. Если данные представляют следующую последовательность чисел:

5.0, 4.24, 2.0, 8.2, 10.32,

то вывод должен выглядеть следующим образом:

2.5, 2.12, 1, 4.1, 5.16

**Уровень 1:**

*Procedure обработка данных;*

ввести данные;

найти минимум введенных данных;

вывести результаты;

*End.*

Детализируем ввод данных.

*Procedure обработка данных;*

определить количество чисел;

*While (пока) не все элементы введены do*

прочитать и запомнить значение элемента;

*End while;*

найти минимум введенных данных;

вывести результаты;

*End.*

Детализируем нахождение минимума.

*Procedure обработка данных;*

определить количество чисел;

*While (пока) не все элементы введены do*

прочитать и запомнить значение элемента;

*End while;*

выбрать в качестве минимума первый элемент данных;

сравнить все значения с минимумом, заменяя

текущий минимум на очередное значение, если оно

меньше его, вывести результаты;

вывести результаты;

*End.*

Вывод результатов можно детализировать следующим образом.

*Procedure обработка данных;*

определить количество чисел;

*While (пока) не все элементы введены do*

прочитать и запомнить значение элемента;

*End while;*

выбрать в качестве минимума первый элемент данных;

сравнить все значения с минимумом, заменяя

текущий минимум на очередное значение, если оно

не превысило его вывести результаты;

*End.*

*While* (пока) не все элементы выведены

вывести значение элемента

*End while;*

**Уровень 2**. Он включает в себя три детализованные выше части, из которых только детализация вторая требует дополнительного внимания. Ее можно детализировать на псевдокоде следующим образом:

*Найти минимум:*

выбрать в качестве минимума первый элемент данных;

*While* (пока) не все элементы проверены;

сравнить все значения с минимумом;

*if* текущее значение меньше минимума *do*

Минимум = текущее значение

*end if*

*End while;*

Задача в приведенном примере проста и не требует разбиения на модули.

При решении реальной задачи может потребоваться написание на псевдокоде многих уровней, чтобы довести все модули до такого состояния, при котором они окажутся готовыми для программирования.

# 3.5 Основы объектно-ориентированного представления программных систем

При объектном подходе модели разрабатываемой системы основываются на предметах и явлениях окружающего мира.

Модель – упрощенное представление реальности. Моделирование необходимо для решения следующих задач:

1) визуализации системы;

2) определения ее структуры и поведения;

3) получения шаблона, позволяющего затем сконструировать систему;

4) документирование принимаемых решений на основании используемой модели.

Для решения этих задач при описании поведения проектируемого программного обеспечения в настоящее время используется UML (Unified Modeling Language) – унифицированный язык моделирования.

## Принципы объектно-ориентированного представления программных систем

В основе объектного подхода к разработке программного обеспечения лежит *объектная декомпозиция*, т. е. представление разрабатываемого программного обеспечения в виде совокупности объектов, в процессе взаимодействия которых через передачу сообщений и происходит выполнение требуемых функций.

В 1995 г. появилась первая версии языка UML, который в настоящее время фактически признан стандартным средством описания проектов, создаваемых с использованием объектно-ориентированного подхода. Его создателями являются ведущие специалисты в этой области: Града Буч, Ивар Якобсон и Джеймс Рамбо.

Спецификация разрабатываемого программного обеспечения при использовании UML объединяет несколько моделей: использования, логическую, реализации, процессов, развертывания (рисунок 3.37).

*Модель использования* представляет собой описание функциональности программного обеспечения с точки зрения пользователя.

*Логическая модель* описывает ключевые абстракции программного обеспечения (классы, интерфейсы и т. п.), т. е. средства, обеспечивающие требуемую функциональность.

*Модель реализации* определяет реальную организацию программных модулей в среде разработки.



Рисунок 3.37 – Полная спецификация разрабатываемого обеспечения при объектном подходе (UML)

*Модель процессов* отображает организацию вычислений и позволяет оценить производительность, масштабируемость и надежность ПО.

*Модель развертывания* показывает особенности размещения программных компонентов на конкретном оборудовании.

Таким образом, каждая из указанных моделей характеризует определенный аспект проектируемой системы, а все они вместе составляют относительно полную модель разрабатываемого программного обеспечения.

Рассмотрение системы любой сложности требует применения техники декомпозиции – разбиения на составляющие элементы.

Объектно-ориентированная декомпозиция обеспечивает разбиение по автономным объектам реального (или виртуального) мира. Эти объекты – более "крупные" элементы, каждый из них несет в себе и описания действий, и описания данных.

Объектно-ориентированное представление ПС основывается на принципах абстрагирования, инкапсуляции, модульности и иерархической организации. Каждый из этих принципов не является новым, но их совместное применение рассчитано на проведение объектно-ориентированной декомпозиции. Это определяет модификацию их содержания и механизмов взаимодействия друг с другом.

### *Абстрагирование*

Аппарат абстракции – инструмент для борьбы со сложностью реальных систем. Создавая понятие в интересах какой-либо задачи, мы отвлекаемся (абстрагируемся) от несущественных характеристик конкретных объектов, определяя только существенные характеристики. Например, в абстракции "часы" мы выделяем характеристику "показывать время", отвлекаясь от таких характеристик конкретных часов, как форма, цвет, материал, цена, изготовитель.

Итак, абстрагирование сводится к формированию абстракций. Каждая абстракция фиксирует основные характеристики объекта, которые отличают его от других видов объектов и обеспечивают ясные понятийные границы.

Абстракция концентрирует внимание на внешнем представлении объекта, позволяет отделить основное в поведении объекта от его реализации. Абстракцию удобно строить путем выделения обязанностей объекта.

### *Инкапсуляция*

Инкапсуляция и абстракция – взаимодополняющие понятия: абстракция выделяет внешнее поведение объекта, а инкапсуляция содержит и скрывает реализацию, которая обеспечивает это поведение. Инкапсуляция достигается с помощью информационной закрытости. Обычно скрываются структура объектов и реализация их методов.

Инкапсуляция является процессом разделения элементов абстракции на секции с различной видимостью. Инкапсуляция служит для отделения интерфейса абстракции от ее реализации.

Пример:физический объект регулятор скорости.

Обязанности регулятора:

* включаться;
* выключаться;
* увеличивать скорость;
* уменьшать скорость;
* отображать свое состояние.

### *Модульность*

В языках C#, С++, Object Pascal, Ada 95 и др. абстракции классов и объектов формируют логическую структуру системы. При производстве физической структуры эти абстракции помещаются в модули. В больших системах, где классов сотни, модули помогают управлять сложностью. Модули служат физическими контейнерами, в которых объявляются классы и объекты логической разработки.

Модульность определяет способность системы подвергаться декомпозиции на ряд сильно связанных и слабо сцепленных модулей.

Общая цель декомпозиции на модули – это уменьшение сроков разработки и стоимости ПС за счет выделения модулей, которые проектируются и изменяются независимо. Каждая модульная структура должна быть достаточно простой, чтобы быть полностью понятой. Изменение реализации модулей должно проводиться без знания реализации других модулей и без влияния на их поведение.

Определение классов и объектов выполняется в ходе логической разработки, а определение модулей – в ходе физической разработки системы. Эти действия сильно взаимосвязаны и осуществляются итеративно.

### *Иерархическая организация*

Рассмотрено три механизма для борьбы со сложностью:

* абстракцию (она упрощает представление физического объекта);
* инкапсуляцию (закрывает детали внутреннего представления абстракций);
* модульность (дает путь группировки логически связанных абстракций).

Дополнением к этим механизмам является иерархическая организация – формирование из абстракций иерархической структуры. Определением иерархии в проекте упрощаются понимание проблем заказчика и их реализация, т.е. сложная система становится обозримой человеком.

Иерархическая организация задает размещение абстракций на различных уровнях описания системы.

Двумя важными инструментами иерархической организации в объектно-ориентированных системах являются:

* структура из классов *("is a"*-иерархия);
* структура из объектов *("part of"*-иерархия).

Чаще всего "*is а*"-иерархическая структура строится с помощью наследования. Наследование определяет отношение между классами, где класс разделяет структуру или поведение, определенные в одном другом (единичное наследование) или в нескольких других (множественное наследование) классах.

Другая разновидность иерархической организации – "*part of*"-иерархическая структура – базируется на отношении агрегации. Агрегация не является понятием, уникальным для объектно-ориентированных систем. Например, любой язык программирования, разрешающий структуры типа "запись", поддерживает агрегацию. Агрегация особенно полезна в сочетании с наследованием:

1. агрегация обеспечивает физическую группировку логически связанной структуры;
2. наследование позволяет легко и многократно использовать эти общие группы в других абстракциях.

### Общая характеристика объектов

Объекты – это конкретные сущности, которые существуют во времени и пространстве.

*Объект*– это конкретное представление абстракции. Объект обладает индивидуальностью, состоянием и поведением. Структура и поведение подобных объектов определены в их общем классе. Термины "экземпляр класса" и "объект" взаимозаменяемы. На рисунке 3.38 приведен пример объекта по имени Стул, имеющего определенный набор свойств и операций.

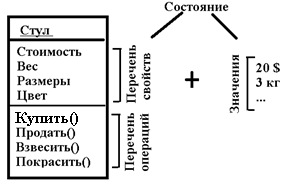


Рисунок 3.38 – Представление объекта с именем Стул

*Индивидуальность –* это характеристика объекта, которая отличает его от всех других объектов.

*Состояние* объекта характеризуется перечнем всех свойств объекта и текущими значениями каждого из этих свойств (рисунок 3.38).

Объекты не существуют изолированно друг от друга. Они подвергаются воздействию или сами воздействуют на другие объекты.

*Поведение* характеризует то, как объект воздействует на другие объекты (или подвергается воздействию) в терминах изменений его состояния и передачи сообщений. Поведение объекта является функцией как его состояния, так и выполняемых им операций (Купить, Продать, Взвесить, Покрасить). Состояние объекта представляет суммарный результат его поведения.

Операция обозначает обслуживание, которое объект предлагает своим клиентам. Возможны пять видов операций клиента над объектом:

1. модификатор (изменяет состояние объекта);
2. селектор (дает доступ к состоянию, но не изменяет его);
3. итератор (доступ к содержанию объекта по частям, в строго определенном порядке);
4. конструктор (создает объект и инициализирует его состояние);
5. деструктор (разрушает объект и освобождает занимаемую им память). Примеры операций приведены в таблице 3.3.

Таблица 3.3 –Разновидности операций

|  |  |
| --- | --- |
| **Вид операции** | **Пример операции** |
| Модификатор | Пополнеть (кг) |
| Селектор | КакойВес ( ) : integer |
| Итератор | ПоказатьАссортиментТоваров ( ) : string |
| Конструктор | СоздатьРобот (параметры) |
| Деструктор | УничтожитьРобот ( ) |

В чистых объектно-ориентированных языках программирования операции могут объявляться только как методы – элементы классов, экземплярами которых являются объекты. Гибридные языки (C++, Ada 95) позволяют писать операции как свободные подпрограммы (вне классов). Соответствующие примеры показаны на рисунке 3.39.

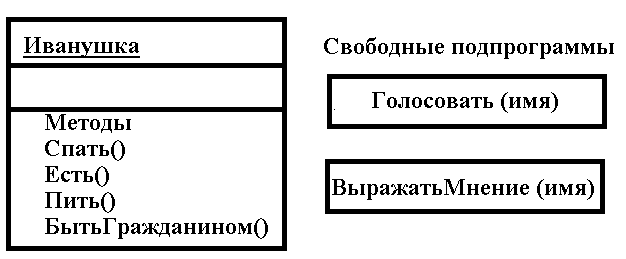
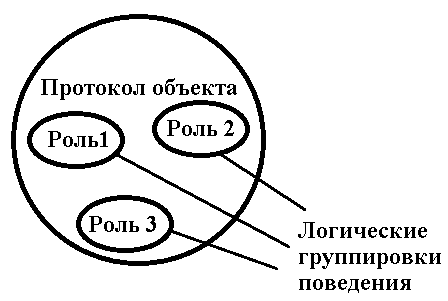


Рисунок 3.39 –Методы и свободные подпрограммы

В общем случае все методы и свободные подпрограммы, ассоциированные с конкретным объектом, образуют его *протокол.* Таким образом, протокол определяет оболочку допустимого поведения объекта и поэтому заключает в себе цельное (статическое и динамическое) представление объекта.

Большой протокол полезно разделять на логические группировки поведения. Эти группировки, разделяющие пространство поведения объекта, обозначают *роли,* которые может играть объект. Выделение ролей приведено на рисунке 3.40.



Рисунке 3.40 –Пространство поведения объекта

С точки зрения внешней среды важное значение имеет понятие – обязанности объекта. *Обязанности* означают обязательства объекта обеспечить определенное поведение. Обязанностями объекта являются все виды обслуживания, которые он предлагает клиентам.

Наличие у объекта внутреннего состояния означает, что важен порядок выполнения им операций. Иначе говоря, объект может представляться как независимый автомат. По аналогии с автоматами можно выделять активные и пассивные объекты (рисунок 3.41).

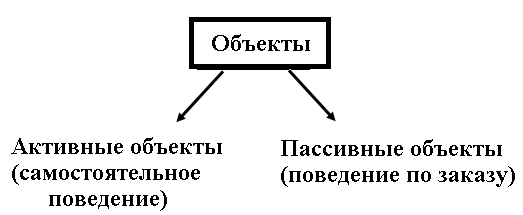


Рисунок 3.41 – Активные и пассивные объекты

Активный объект имеет собственный канал (поток) управления, пассивный – нет. Активный объект автономен, он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, наоборот, может изменять свое состояние только под воздействием других объектов.

### *Виды отношений между объектами*

У разработчика ПО находятся не объекты-одиночки, а взаимодействующие объекты, именно взаимодействие объектов реализует поведение системы. У Г. Буча в книге"Объектно-ориентированный проектирование" есть цитата из Галла: "Самолет – это набор элементов, каждый из которых по своей природе стремится упасть на землю, но ценой совместных непрерывных усилий преодолевает эту тенденцию". Отношения между парой объектов основываются на взаимной информации о разрешенных операциях и ожидаемом поведении. Особо интересны два вида отношений между объектами: связи и агрегация.

### *Связи*

Связь – это физическое или понятийное соединение между объектами. Объект сотрудничает с другими объектами через соединяющие их связи. Связь обозначает соединение, с помощью которого:

* объект-клиент вызывает операции объекта-поставщика;
* один объект перемещает данные к другому объекту.

Связи между объектами показаны на рисунок 3.42 с помощью соединительных линий. Связи представляют возможные пути для передачи сообщений. Сами сообщения показаны стрелками, отмечающими их направления, и помечены именами вызываемых операций.

Как участник связи объект может играть одну из трех ролей:

* актер – объект, который может воздействовать на другие объекты, но никогда не подвержен воздействию других объектов;
* сервер – объект, который никогда не воздействует на другие объекты, он только используется другими объектами;
* агент – объект, который может как воздействовать на другие объекты, так и использоваться ими. Агент создается для выполнения работы от имени актера или другого агента.

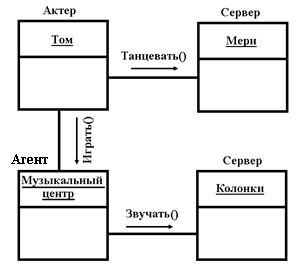


Рисунок 3.42 –Связи между объектами

На рисунке 3.42 Том – это актер, Мери, Колонки – серверы, Музыкальный центр – агент.

### *Видимость объектов*

Рассмотрим два объекта, А и В, между которыми имеется связь. Для того чтобы объект А мог послать сообщение в объект В, надо, чтобы В был виден для А.

Различают четыре формы видимости между объектами:

* объект-поставщик (сервер) глобален для клиента;
* объект-поставщик (сервер) является параметром операции клиента;
* объект-поставщик (сервер) является частью объекта-клиента;
* объект-поставщик (сервер) является локально объявленным объектом в операции клиента.

На этапе анализа вопросы видимости обычно опускают, а на этапах проектирования и реализации рассматриваются вопросы видимости по связям.

### *Агрегация*

Связи обозначают равноправные (клиент-серверные) отношения между объектами. Агрегация обозначает отношения объектов в иерархии "целое/часть". Агрегация обеспечивает возможность перемещения от целого (агрегата) к его частям (свойствам).

Агрегация может обозначать, а может и не обозначать физическое включение части в целое. На рисунке 3.43 приведен пример физического включения (композиции) частей (Двигателя, Сидений, Колес) в агрегат Автомобиль. В этом случае говорят, что части включены в агрегат по величине.

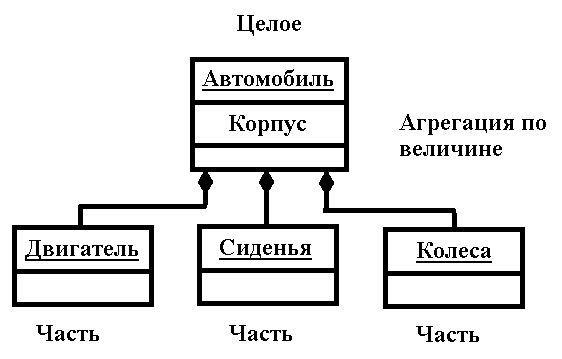


Рисунок 3.43 –Физическое включение частей в агрегат

На рисунке 3.44 приведен пример нефизического включения частей (Студента, Преподавателя) в агрегат ВУЗ. Очевидно, что Студент и Преподаватель являются элементами ВУЗа, но они не входят в него физически. В этом случае говорят, что части включены в агрегат по ссылке.

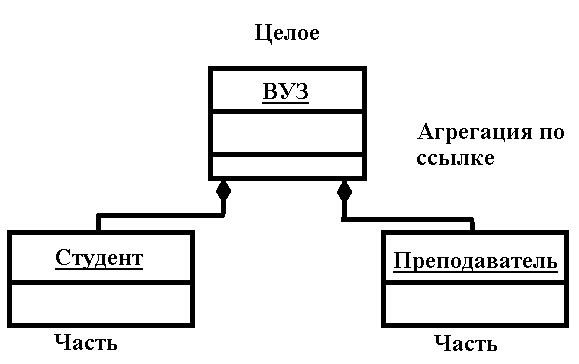


Рисунок 3.44 –Нефизическое включение частей в агрегат

Итак, между объектами существуют два вида отношений – связи и агрегация. Какое из них выбрать?

При выборе вида отношения должны учитываться следующие факторы:

* связи обеспечивают низкое сцепление между объектами;
* агрегация инкапсулирует части как единое целое.

## Классы

Понятия объекта и класса тесно связаны. Тем не менее существует важное различие между этими понятиями. Класс – это абстракция существенных характеристик объекта.

Класс– описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Любой объект – просто экземпляр класса.

Как показано на рисунок 3.45, различают внутреннее представление класса (реализацию) и внешнее представление класса (интерфейс).



Рисунок 3.45 –Структуре представления класса

Интерфейс объявляет возможности (услуги) класса, но скрывает его структуру и поведение. Он демонстрирует внешнему миру абстракцию класса, его внешний облик. Интерфейс в основном состоит из объявлений всех операций, применимых к экземплярам класса. Он может также включать объявления типов, переменных, констант и исключений, необходимых для полноты данной абстракции.

Интерфейс может быть разделен на 3 части:

1. публичную *(public),* объявления которой доступны всем клиентам;
2. защищенную *(protected),* объявления которой доступны только самому классу, его подклассам и друзьям;
3. приватную *(private),* объявления которой доступны только самому классу и его друзьям.

Другом класса называют класс, который имеет доступ ко всем частям этого класса (публичной, защищенной и приватной). Иными словами, от друга у класса нет секретов. Другом класса может быть и свободная программа. Реализация класса описывает секреты поведения класса. Она включает реализации всех операций, определенных в интерфейсе класса.

### *Виды отношений между классами*

Классы, подобно объектам, не существуют в изоляции. Напротив, с отдельной проблемной областью связывают ключевые абстракции, отношения между которыми формируют структуру из классов системы.

Всего существует четыре основных вида отношений между классами:

* ассоциация (фиксирует структурные отношения – связи между экземплярами классов);
* зависимость (отображает влияние одного класса на другой класс);
* обобщение-специализация ("*is а*"-отношение);
* целое-часть ("*part of*"-отношение).

Для покрытия основных отношений большинство объектно-ориентированных языков программирования поддерживает следующие отношения:

1) ассоциация;

2) наследование;

3) агрегация;

4) зависимость;

5) конкретизация;

6) метакласс;

7) реализация.

*Ассоциации* обеспечивают взаимодействия объектов, принадлежащих разным классам. Они являются связующим, соединяющим воедино все элементы программной системы. Благодаря ассоциациям, мы получаем работающую систему. Без ассоциаций система превращается в набор изолированных классов-одиночек.

Ассоциация обозначает семантическое соединение классов.

Пример:в системе обслуживания читателей имеются две ключевые абстракции – Книга и Библиотека. Класс Книга играет роль элемента, хранимого в библиотеке. Класс Библиотека играет роль хранилища для книг (рисунок 3.46).

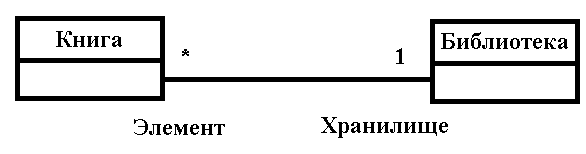


Рисунок 3.46 – Ассоциация

Отношение ассоциации между классами изображено на рисунке 3.46. Очевидно, что ассоциация предполагает двухсторонние отношения:

* для данного экземпляра Книги выделяется экземпляр Библиотеки, обеспечивающий ее хранение;
* для данного экземпляра Библиотеки выделяются все хранимые Книги.

Здесь показана ассоциация *один-ко-многим.* Каждый экземпляр Книги имеет указатель на экземпляр Библиотеки. Каждый экземпляр Библиотеки имеет набор указателей на несколько экземпляров Книги.

Ассоциация обозначает только семантическую связь. Она не указывает направление и точную реализацию отношения. Ассоциация пригодна для анализа проблемы, когда требуется идентифицировать связи. Ассоциация *один-ко-многим,* введенная в примере, означает, что для каждого экземпляра класса Библиотека есть *0* или более экземпляров класса Книга, а для каждого экземпляра класса Книга есть один экземпляр Библиотеки. Эту множественность обозначает *мощность ассоциации.* Мощность ассоциации бывает одного из трех типов:

* один-к-одному;
* один-ко-многим;
* многие-ко-многим.

Примеры ассоциаций с различными типами мощности приведены на рисунке 3.47, они имеют следующий смысл:

* у европейской жены один муж, а у европейского мужа одна жена;
* у восточной жены один муж, а у восточного мужа сколько угодно жен;
* у заказа один клиент, а у клиента сколько угодно заказов;
* человек может посещать сколько угодно зданий, а в здании может находиться сколько угодно людей.

*Наследование* – наиболее популярная разновидность отношения *обобщение-специализация.* Альтернативой наследованию считается делегирование. При делегировании объекты *делегируют* свое поведение родственным объектам. В этом случае классы становятся не нужны.

Наследование – это отношение, при котором один класс разделяет структуру и поведение, определенные в одном другом (простое наследование) или во многих других (множественное наследование) классах.

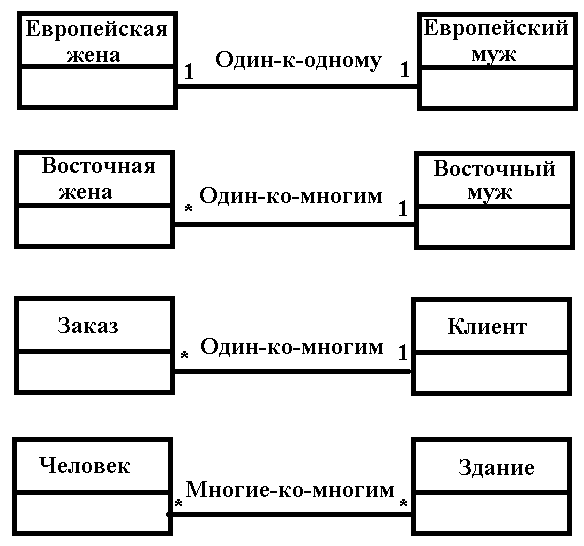


Рисунок 3.47 –Ассоциации с различными типами мощности

Между *п* классами наследование определяет иерархию *"является" ("is а")*, при которой подкласс наследует от одного или нескольких более общих суперклассов. Говорят, что подкласс является специализацией его суперкласса (за счет дополнения или переопределения существующей структуры или поведения).

*Обобщением* называют такое отношение между классами, при котором любой объект одного класса (подтипа) обязательно является также и объектом другого класса, называемого в данном контексте *супертипом*. Так, если некоторый конкретный студент Иванов И.И. является объектом подтипа Студент первого курса супертипа Студент, то тот же самый Иванов И.И. является объектом указанного супертипа. Следовательно, все, что известно об объектах супертипа (ассоциации, атрибуты, операции), касается и объектов подтипа. На диаграмме классов обобщение обозначают линией с треугольной стрелкой на конце, направленной к супертипу (рисунок 3.48).

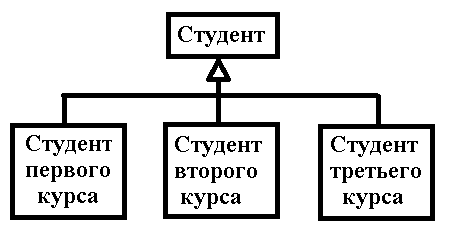


Рисунок 3.48 – Обозначение обобщения

На практике определение основных понятий предметной области, которые должны представляться на контекстной диаграмме в виде классов, является не тривиальной задачей.

Обычно используют следующий способ:

* формируют множество понятий-кандидатов из существительных, характеризующих предметную область в описании вариантов использования;
* исключают понятия, не существенные для данного варианта использования.

*Агрегация* обеспечивает отношения *целое-часть,* объявляемые для экземпляров классов.

Отношения агрегации между классами аналогичны отношениям агрегации между объектами.

Графическая иллюстрация отношения агрегации по величине (композиции) представлена на рисунке 3.49.

Возможен косвенный тип агрегации – включением по ссылке (рисунок 3.49).

Здесь показаны класс-агрегат Дом и класс-агрегат Окно, причем указаны роли и множественность частей агрегата (соответствующие пометки имеют линии отношений).

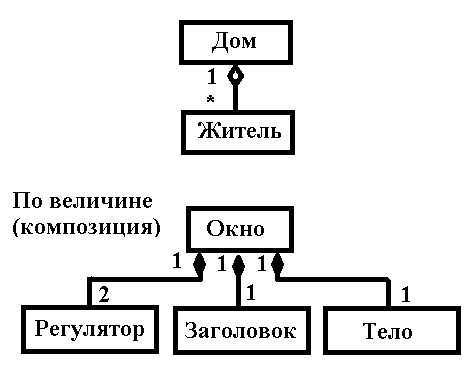


Рисунок 3. 49 –Агрегация классов

Возможны и другие формы представления агрегации по величине – композиции (рисунок 3.50). Композицию можно отобразить графическим вложением символов частей в символ агрегата (левая часть рисунка 3.50). Вложенные части демонстрируют свою множественность (мощность, кратность) в правом верхнем углу своего символа. Если метка множественности опущена, по умолчанию считают, что ее значение "много". Вложенный элемент может иметь роль в агрегате. Используется синтаксис роль : имяКласса.

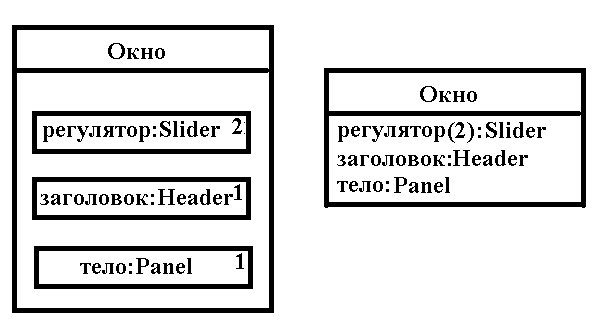


Рисунок 3.50 –Формы представления композиции

Эта роль соответствует той роли, которую играет часть в неявном (в этой нотации) отношении композиции между частью и целым (агрегатом).

В правой части рисунка 3.50 представлены свойства (атрибуты) класса, находящиеся в отношении композиции между всем классом и его элементами-свойствами. В общем случае свойства должны иметь примитивные значения (числа, строки, даты), а не ссылаться на другие классы, так как в "атрибутной" нотации не видны другие отношения классов-частей. Кроме того, свойства классов не могут находиться в совместном использовании несколькими классами.

*Зависимость* часто представляется в виде частной формы – *использования,* которое фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.

Зависимость – это отношение, которое показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый), который его использует. Графически зависимость изображается как пунктирная стрелка, направленная на класс, от которого зависит. С помощью зависимости уточняют, какая абстракция является клиентом, а какая – поставщиком определенной услуги. Пунктирная стрелка зависимости направлена от клиента к поставщику.

Наиболее часто зависимости показывают, что один класс использует другой класс как аргумент в сигнатуре своей операции.

*Конкретизация* выражает другую разновидность отношения *обобщение-специализация.* Применяется в таких языках, как Ada 95, C++, Эйфель.

Г. Буч определяет конкретизацию как процесс наполнения шаблона (родового или параметризованного класса). Целью является получение класса, от которого возможно создание экземпляров.

Родовой класс служит заготовкой, шаблоном, параметры которого могут наполняться (настраиваться) другими классами, типами, объектами, операциями. Он может быть родоначальником большого количества обычных (конкретных) классов. Возможности настройки родового класса представляются списком формальных родовых параметров. Эти параметры в процессе настройки должны заменяться фактическими родовыми параметрами. Процесс настройки родового класса называют конкретизацией.

В разных языках программирования родовые классы оформляются по-разному.

*Метакласс* – это класс классов, понятие, позволяющее обращаться с классами как с объектами.

*Реализация* определяет отношение, при котором класс-приемник обеспечивает свою собственную реализацию интерфейса другого класса-источника. Иными словами, здесь идет речь о наследовании интерфейса. Семантически реализация – это "скрещивание" отношений зависимости и обобщения-специализации.

*Полиморфизм* – возможность с помощью одного имени обозначать операции из различных классов (но относящихся к общему суперклассу). Вызов обслуживания по полиморфному имени приводит к исполнению одной из некоторого набора операций.

**3.6 Архитектура программного обеспечения**

В литературе приводятся разные определения архитектуры ПС.

*Архитектурой программного обеспечения* называют совокупность базовых концепций (принципов) его построения.

*Архитектура ПС* – это его строение, как оно видно (или должно быть видно) извне его, т. е. представление ПС как системы, состоящей из некоторой совокупности взаимодействующих подсистем.

*Архитектура* – это структура организации и связанное с ней поведение системы. Архитектуру можно рекурсивно разобрать на части, взаимодействующие посредством интерфейсов, связи, которые соединяют части, и условия сборки частей. Части, которые взаимодействуют через интерфейсы, включают классы, компоненты и подсистемы.

Выбор архитектуры разрабатываемого ПО определяется задачами, поставленными перед разработчиками, функциональными и эксплуатационными требованиями.

С точки зрения количества пользователей, работающих с одной копией ПО,различают:

* однопользовательскую архитектуру;
* многопользовательскую (сетевую) архитектуру.

Кроме того, в рамках однопользовательской архитектуры различают:

* программы. Программа (program, routine) – упорядоченная последовательность формализованных инструкций для решения задачи с помощью компьютера. Это самый простой вид архитектуры, который обычно используется при решении небольших задач;
* пакеты программ. Пакеты программ представляют собой несколько отдельных программ, решающих задачи определенной прикладной области. Например, пакет графических программ, пакет математических программ. Пакет программ реализуется как набор отдельных программ, каждая из которых сама вводит необходимые данные и выводит результаты, т. е. программы пакета связаны между собой только принадлежностью к некоторой прикладной области;
* программные комплексы. Программные комплексы представляют собой совокупность программ, совместно обеспечивающих решение небольшого класса сложных задач одной прикладной области. При этом для выполнения некоторой задачи программой-диспетчером последовательно вызываются несколько программ из программного комплекса. Поскольку несколько программ для решения одной задачи работают с одними и теми же исходными данными и промежуточными результатами, желательно хранить эти данные и результаты вызовов в оперативной памяти или в файлах в пределах одного пользовательского проекта. Программы комплекса могут компилироваться как самостоятельные единицы или совместно. Программа-диспетчер может иметь примитивный интерфейс и простую справочную систему;
* программные системы. Программные системы представляют собой организованную совокупность программ (подсистем), позволяющую решать широкий класс задач из некоторой прикладной области. Программы, входящие в программную систему, взаимодействуют через общие данные. Программные системы имеют достаточно развитый интерфейс, что требует их тщательного проектирования и разработки.

Многопользовательскую архитектуру реализуют системы, построенные по принципу "клиент – сервер".

### Анализ области решений, компонентный подход

Целью этой деятельности является понимание, можно ли вообще решить стоящие перед разрабатываемой системой задачи, при каких условиях и ограничениях это можно сделать, как они решаются, если решение есть, а если нет – нельзя ли придумать способ его найти или получить хотя бы приблизительное решение, и т.п. Когда определены принципиальные способы решения всех поставленных задач (быть может, в каких-то ограничениях), основной проблемой становится способ организации программной системы, который позволил бы реализовать все эти решения и при этом удовлетворить требованиям, касающимся нефункциональных аспектов разрабатываемой программы. Искомый способ организации ПО в виде системы взаимодействующих компонентов называют *архитектурой*, а процесс ее создания – *проектированием архитектуры ПО*.

Под *компонентом* в этом определении имеется в виду достаточно произвольный структурный элемент ПО, который можно выделить, определив интерфейс взаимодействия между этим компонентом и всем, что его окружает. Обычно при разработке ПО термин "*компонент*" имеет несколько другой, более узкий смысл – это единица развертывания, самая маленькая часть системы, которую можно включить или не включить в ее состав. Такой компонент также имеет определенный интерфейс и удовлетворяет некоторому набору правил.

В определении архитектуры упоминается набор структур, а не одна структура. Это означает, что в качестве различных аспектов архитектуры, различных взглядов на нее выделяются различные структуры, соответствующие разным аспектам взаимодействия компонентов. Примеры таких аспектов – описание типов компонентов и типов статических связей между ними при помощи диаграмм классов, описание композиции компонентов при помощи структур ссылающихся друг на друга объектов, описание поведения компонентов при помощи моделирования их как набора взаимодействующих, передающих друг другу некоторые события, конечных автоматов.

Архитектура программной системы похожа на набор карт некоторой территории. Карты имеют разные масштабы, на них показаны разные элементы (административно-политическое деление, рельеф и тип местности – лес, степь, пустыня, болота и пр., экономическая деятельность и связи), но они объединяются тем, что все представленные на них сведения соотносятся с географическим положением. Точно так же архитектура ПО представляет собой набор структур или представлений, имеющих различные уровни абстракции и показывающих разные аспекты (структуру классов ПО, структуру развертывания, т.е. привязки компонентов ПО к физическим машинам, возможные сценарии взаимодействий компонентов и пр.), объединяемых сопоставлением всех представленных данных со структурными элементами ПО.

Архитектура определяет большинство характеристик качества ПО в целом. Архитектура служит также основным средством общения между разработчиками, а также между разработчиками и всеми остальными лицами, заинтересованными в данном ПО.

Выбор архитектуры задает способ реализации требований на высоком уровне абстракции. Именно архитектура почти полностью определяет такие характеристики ПО, как надежность, переносимость и удобство сопровождения. Она также значительно влияет на удобство использования и эффективность ПО, которые сильно зависят и от реализации отдельных компонентов. Значительно меньше влияние архитектуры на функциональность – обычно заданную функциональность можно реализовать, использовав совершенно различные архитектуры.

Поэтому выбор между той или иной архитектурой определяется в большей степени именно нефункциональными требованиями и необходимыми свойствами ПО с точки зрения удобства сопровождения и переносимости. При этом для построения хорошей архитектуры надо учитывать возможные противоречия между требованиями к различным характеристикам и уметь выбирать компромиссные решения, дающие приемлемые значения по всем показателям.

Стандартами, регламентирующими описание архитектуры, которые являются основной составляющей проектной документации на ПО, это IEEE 1016-1998 (Recommended Practice for Software Design Descriptions – рекомендуемые методы описаний проектных решений для ПО) и IEEE 1471-2000 (Recommended Practice for Architectural Description of Software-Intensive Systems – рекомендуемые методы описания архитектуры программных систем).

Стандарт IEEE 1471 определяет представление архитектуры(architectural description) как согласованный набор документов, описывающий архитектуру с точки зрения определенной группы заинтересованных лиц с помощью набора моделей. Архитектура может иметь несколько представлений, отражающих интересы различных групп заинтересованных лиц.

Стандарт IEEE 1471 отмечает необходимость использования архитектуры системы для решения следующих задач:

* анализ альтернативных проектов системы;
* планирование изменений в проектирование системы, внесения изменений в ее организацию;
* общение по поводу системы между различными организациями, вовлеченными в ее разработку, эксплуатацию, сопровождение, приобретение, продажу;
* определение критериев приемки системы при ее сдаче в эксплуатацию;
* разработка документации по ее использованию и сопровождению, включая обучающие и маркетинговые материалы;
* проектирование и разработка отдельных элементов системы;
* сопровождение, эксплуатация, управление конфигурациями и внесение изменений и поправок;
* планирование бюджета и использования других ресурсов в проектах, связанных с разработкой, сопровождением или эксплуатацией системы;
* анализ и оценка качества системы.

**Разработка и оценка архитектуры на основе сценариев**

При проектировании архитектуры системы в виде вариантов использования, первые возможные шаги состоят в следующем.

1. Выделение компонентов:

* выбирается набор "основных" сценариев использования – наиболее существенных и выполняемых чаще других;
* исходя из опыта проектировщиков, выбранного архитектурного стиля и требований к переносимости и удобству сопровождения системы определяются компоненты, отвечающие за определенные действия в рамках этих сценариев, т.е. за решение определенных подзадач;
* каждый сценарий использования системы представляется в виде последовательности обмена сообщениями между полученными компонентами;
* при возникновении дополнительных хорошо выделенных подзадач добавляются новые компоненты, и сценарии уточняются.

2. Определение интерфейсных компонентов:

* для каждого компонента выделяется его интерфейс – набор сообщений, которые он принимает и посылает другим компонентам;
* рассматриваются второстепенные сценарии, которые так же разбиваются на последовательности обмена сообщениями;
* если интерфейсный компонент многофункционален он разбивается на более мелкие составляющие.

3. Уточнение набора компонентов

4. Достижение нужных свойств. Должны быть выполнены следующие условия:

* все сценарии использования реализуются в виде последовательностей обмена сообщениями между компонентами в интерфейсах;
* набор компонентов достаточен для обеспечения всей нужной функциональности, удобен для сопровождения или портирования на другие платформы и не вызывает заметных проблем производительности;
* каждый компонент имеет небольшой и четко очерченный круг решаемых задач и строго определенный, сбалансированный по размеру интерфейс.

На основе возможных сценариев использования или модификации системы возможен также анализ характеристик архитектуры и оценка ее пригодности для поставленных задач или сравнительный анализ нескольких архитектур. Это метод анализа архитектуры ПО (Software Architecture Analysis Method, SAAM).

Основные шаги SAAM:

* определить набор сценариев действий пользователей или внешних систем, которые могут планироваться для реализации или быть новыми;
* определить архитектуру (или несколько сравниваемых архитектур);
* классифицировать сценарии. Для каждого сценария определить поддерживается ли он данной архитектурой или для его поддержки нужно вносить изменения;
* оценить сценарии. Определить, какие сценарии полностью поддерживаются рассматриваемыми архитектурами. Для неподдерживаемого сценария необходимо определить изменения в архитектуре – внесение новых компонентов, изменения в существующих, изменения связей и способов взаимодействия;
* выявить взаимодействие сценариев. Определить какие компоненты требуется изменять для неподдерживаемых сценариев. Если требуется изменять один компонент для поддержки нескольких сценариев, то такие сценарии называют взаимодействующими;
* оценить архитектуру в целом (или сравнить несколько заданных архитектур). Для этого используют оценки важности сценариев и степень их поддержки архитектурой.

### 3.7 Диаграммы UML

На настоящий момент наиболее проработанным и наиболее широко используемым является *унифицированный язык моделирования* (*Unified Modeling Language, UML*).

 UML – стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов. UML – это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 95, Object Pascal и др.) и даже в таблицы для реляционной БД.

Для представления архитектуры, а точнее – различных входящих в нее структур, удобно использовать графические языки, хотя достаточно часто архитектуру системы описывают просто набором именованных прямоугольников, соединенных линиями и стрелками, которые представляют возможные связи.

UML предлагает использовать для описания архитектуры восемь видов диаграмм. Девятый вид UML диаграмм – диаграммы вариантов использования, не относится к архитектурным представлениям. Другие виды диаграмм можно использовать для описания внутренней структуры компонентов или сценариев действий пользователей, также не относящихся к архитектуре. Диаграммы UML делятся на две группы – *статические* и *динамические диаграммы*.

Статические диаграммы представляют либо постоянно присутствующие в системе сущности и связи между ними, либо суммарную информацию о сущностях и связях, либо сущности и связи, существующие в какой-то определенный момент времени. Они не показывают способов поведения этих сущностей. К этому типу относятся *диаграммы классов*, *объектов*, *компонентов* и *диаграммы развертывания*.

Динамические диаграммы описывают происходящие в системе процессы. К ним относятся *диаграммы деятельности*, *сценариев*, *диаграммы* *взаимодействия* и *диаграммы состояний*.

**UML**

Словарь UML образуют три разновидности строительных блоков: *предметы, отношения, диаграммы.*

*Предметы* – это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

В UML имеются четыре разновидности предметов:

* структурные предметы;
* предметы поведения;
* группирующие предметы;
* поясняющие предметы.

Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для написания моделей.

*Структурные предметы* являются существительными в UML-моделях. Они представляют статические части модели – понятийные или физические элементы. Используют восемь разновидностей структурных предметов.

1. *Класс* – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рисунке 3.51, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями (см. раздел 3.5).

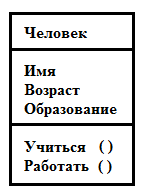


Рисунок 3.51 – Классы

1. *Интерфейс –* набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне. Интерфейс может представлять полные услуги класса или компонента или часть таких услуг. Интерфейс определяет набор спецификаций операций (их сигнатуры), а не набор реализаций операций. Графически интерфейс изображается в виде кружка с именем, как показано на рисунке 3.52. Имя интерфейса обычно начинается с буквы "I". Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.



Рисунок 3.52 – Интерфейсы

1. *Кооперация* (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов. Таким образом, кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют реализацию паттернов (образцов), которые формируют систему. Как показано на рисунок 3.53, графически кооперация изображается как пунктирный эллипс, в который вписывается ее имя.

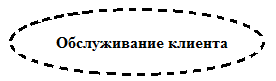


Рисунок 3.53 – Кооперация

1. *Актер –* набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case). Каждая роль требует от системы определенного поведения. Как показано на рисунке 3.54, актер изображается как проволочный человечек с именем.

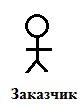


Рисунок 3.54 – Актеры

1. *Элемент Use Case* (Прецедент) – описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актера и производящих видимый для актера результат. В модели элемент Use Case применяется для структурирования предметов поведения. Элемент Use Case реализуется кооперацией. Как показано на рисунке 3.55, элемент Use Case изображается как эллипс, в который вписывается его имя.



Рисунок 3.55 –Элементы Use Case

1. *Активный класс –* класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность. Активный класс похож на обычный класс за исключением того, что его объекты действуют одновременно с объектами других классов. Как показано на рисунке 3.56, активный класс изображается как утолщенный прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.

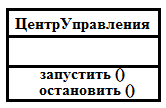


Рисунок 3.56 –Активные классы

1. *Компонент –* физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. В систему включаются как компоненты, являющиеся результатами процесса разработки (файлы исходного кода), так и различные разновидности используемых компонентов (СОМ+ -компоненты, Java Beans). Обычно компонент – это физическая упаковка различных логических элементов (классов, интерфейсов и сотрудничеств). Как показано на рисунке 3.57, компонент изображается как прямоугольник с вкладками, обычно включающий имя.

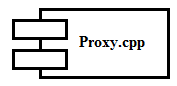


Рисунок 3.57 –Компоненты

1. *Узел –* физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Как показано на рисунке 3.58, узел изображается как куб с именем.



Рисунок 3.58 –Узлы

*Предметы поведения –* динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существует две основные разновидности предметов поведения.

1. *Взаимодействие* – поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели. Взаимодействие может определять динамику, как совокупности объектов, так и отдельной операции. Элементами взаимодействия являются сообщения, последовательность действий (поведение, вызываемое сообщением) и связи (соединения между объектами). Как показано на рисунке 3.59, сообщение изображается в виде направленной линии с именем ее операции.



Рисунок 3.59 – Сообщения

1. *Конечный автомат –* поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события (и с учетом обязанностей по этим событиям). С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход). Как показано на рисунке 3.60, состояние изображается как закругленный прямоугольник, обычно включающий его имя и его подсостояния (если они есть).



Рисунок 3.60 –Состояния

Эти два элемента – взаимодействия и конечные автоматы – являются базисными предметами поведения, которые могут включаться в UML-модели. Семантически эти элементы ассоциируются с различными структурными элементами (прежде всего с классами, сотрудничествами и объектами).

*Группирующие предметы* – организационные части UML-моделей. Это ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета – пакет.

*Пакет –* общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет – чисто концептуальное понятие. Это означает, что пакет существует только в период разработки. Как показано на рисунке 3.61, пакет изображается как папка с закладкой, на которой обозначено его имя и, иногда, его содержание.

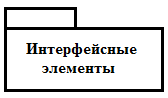


Рисунок 3.61 –Пакеты

*Поясняющие предметы –* разъясняющие части UML-моделей. Они являются замечаниями, которые можно применить для описания, объяснения и комментирования любого элемента модели. Предусмотрена одна разновидность поясняющего предмета – примечание.

*Примечание –* символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов. Как показано на рисунке 3.62, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.

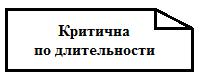


Рисунок 3.62 –Примечания

## Отношения в UML

В UML имеются четыре разновидности отношений:

1) зависимость;

2) ассоциация;

3) обобщение;

4) реализация.

Эти отношения являются базовыми строительными блоками отношений. Они используются при написании моделей.

1. *Зависимость* – семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета). Как показано на рисунке 3.63, зависимость изображается в виде пунктирной линии, возможно направленной на независимый предмет и иногда имеющей метку.



Рисунок 3.63 –Зависимости

1. *Ассоциация –* структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация – это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рисунке 3.64, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей мощность и имена ролей.



Рисунок 3.64 –Ассоциации

1. *Обобщение* – отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рисунке 3.65, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.



Рисунок 3.65 –Обобщения

1. *Реализация –* семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации). Отношения реализации применяют в двух случаях: между интерфейсами и классами (или компонентами), реализующими их; между элементами Use Case и кооперациями, которые их реализуют. Реализация изображается как нечто среднее между обобщением и зависимостью (рисунок 3.66).



Рисунок 3.66 –Реализации

## Диаграммы

*Диаграмма* – графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма дает неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. Теоретически диаграмма может содержать любую комбинацию предметов и отношений, на практике ограничиваются малым количеством комбинаций, которые соответствуют пяти представлениям архитектуры ПС. По этой причине UML включает девять видов диаграмм:

1) диаграммы классов;

2) диаграммы объектов (пакетов);

3) диаграммы Use Case (диаграммы прецедентов или вариантов использования);

4) диаграммы последовательности;

5) диаграммы сотрудничества (кооперации);

6) диаграммы состояний;

7) диаграммы деятельности;

8) компонентные диаграммы;

9) диаграммы размещения (развертывания).

***1. Диаграмма классов***показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Диаграммы классов обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

* концептуальный уровень, на котором диаграммы классов отображают связи между основными понятиями предметной области;
* уровень спецификаций, на котором диаграммы классов отображают связи объектов этих классов;
* уровень реализации, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

Каждую из перечисленных моделей используют на конкретном этапе разработки программного обеспечения:

* концептуальную модель – на этапе анализа;
* диаграммы классов уровня спецификации – на этапе проектирования;
* диаграммы классов уровня реализации – на этапе реализации.

Диаграмма классов может отражать различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. Диаграммы классов обычно содержат следующие сущности:

* классы;
* интерфейсы;
* кооперации;
* отношения зависимости, обобщения и ассоциации.

Когда говорят о данной диаграмме, имеют в виду статическую структурную модель проектируемой системы, поэтому диаграмму классов принято считать графическим представлением таких взаимосвязей логической модели системы, которые не зависят от времени.

*Класс* (class) в языке UML служит для обозначения множества объектов, которые имеют одинаковую структуру, поведение и отношения с объектами из других классов. На диаграмме класс изображают в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на две или три секции (рисунок 3.67). В этих разделах могут указываться имя класса, атрибуты (переменные) и операции (методы). Иногда в графическом изображении класса добавляется четвертая секция, содержащая описание исключительных ситуаций.

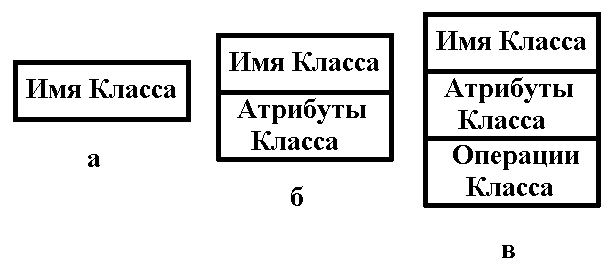


Рисунок 3.67 – Графическое изображение класса на диаграмме классов

Обязательным элементом обозначения класса является его имя. На начальных этапах разработки диаграммы класс может обозначаться простым прямоугольником с указанием только его имени (рис. 3.67, а). В дальнейшем диаграммы описания классов дополняются атрибутами (рис. 3.67, б) и операциями (рисунок 3.67, в).

Чтобы сразу отличить класс от других элементов языка UML, секцию атрибутов и операций выделяют горизонтальной линией, даже если она является пустой. На рисунке 3.68 приведены примеры графического изображения классов на диаграмме классов. В первом случае для класса "Прямоугольник" (рисунок 3.68, а) указаны только его атрибуты – точки на координатной плоскости, определяющие его местоположение.

Для класса "Окно" (рисунок 3.68, б) указаны только его операции (показать(), скрыть()), секция атрибутов оставлена пустой. Для класса "Счет" (рисунок 3.68, в), кроме операции проверки кредитной карточки, дополнительно изображена четвертая секция, в которой указано исключение – отказ от обслуживания просроченной кредитной карточки.

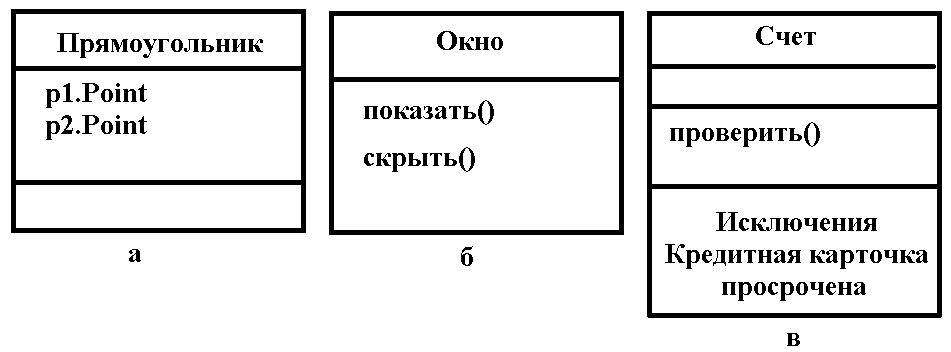


Рисунок 3.68 – Примеры графического изображения классов на диаграмме

Имя класса должно быть уникальным в пределах диаграммы или совокупности диаграмм классов пакета. Оно указывается в первой верхней секции прямоугольника, записывается по центру секции имени полужирным шрифтом и должно начинаться с заглавной буквы. В качестве имен классов рекомендуется использовать одно или несколько существительных без пробелов между ними. Кроме того, в секции обозначения класса могут находиться ссылки на стандартные шаблоны или абстрактные классы, от которых образован данный класс и, соответственно, от которых он наследует свойства и методы.

Примерами имен классов могут быть такие существительные, как "Сотрудник", "Фирма", "Руководитель", "Покупатель", "Продавец" и многие другие, имеющие непосредственное отношение к моделируемой предметной области и функциональному назначению проектируемой системы.

Если класс не имеет экземпляров (объектов), то он называется абстрактным классом, его имя записывается курсивом так же, как любой текст, относящийся к абстрактному элементу.

Во второй секции прямоугольника – графического изображения класса – записываются его атрибуты (attributes) или свойства. Стандартная запись атрибута в языке UML выглядит следующим образом:

*<квантор видимости><имя атрибута>[кратность]*

*<тип атрибута> <исходное значение>{строка-свойство}*

Квантор видимости может быть опущен – это означает, что видимость атрибута не указывается либо же должна принимать одно из трех возможных значений:

* общедоступный (*public*) – обозначается "+";
* защищенный (*protected*) – обозначается "#";
* закрытый (*private*) – обозначается "-".

*Имя атрибута* – единственный обязательный элемент обозначения атрибута, представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и является уникальной в пределах данного класса.

Кратность атрибута показывает количество конкретных атрибутов данного типа, входящих в состав класса, и обозначается следующим образом:

[*нижняя\_граница1 верхняя\_граница1, нижняя\_граница2,*

*верхняя\_граница2, нижняя\_граница n верхняя\_граница n],*

где *нижняя\_граница* и *верхняя\_граница* являются положительными целыми числами, каждая пара которых служит для обозначения отдельного замкнутого интервала целых чисел. В качестве верхней границы может использоваться специальный символ "\*", который означает произвольное положительное целое число, т. е. неограниченное сверху значение кратности соответствующего атрибута.

Если в качестве кратности указывается единственное число, то кратность атрибута принимается равной данному числу. Значения кратности из интервала следуют в монотонно возрастающем порядке без пропуска отдельных чисел, лежащих между нижней и верхней границами, соответствующие нижние и верхние границы интервалов включаются в значение кратности. Если же указывается единственный знак "\*", то это означает, что кратность атрибута может быть произвольным положительным целым числом или нулем.

Могут использоваться следующие варианты задания кратности атрибутов:

* [0..1] означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие значения для данного атрибута;
* [0..\*] или просто [\*] означает, что кратность атрибута может принимать любое положительное целое значение, большее или равное 0;
* [1..\*] означает, что кратность атрибута может принимать любое положительное целое значение, большее или равное 1;
* [1..5] означает, что кратность атрибута может принимать любое значение из чисел 1, 2, 3, 4, 5;
* [1..3,7.. 10] означает, что кратность атрибута может принимать любое значение из чисел 1, 2, 3, 7, 8, 9, 10;
* [1..3,7..\*] означает, что кратность атрибута может принимать любое значение из чисел 1, 2, 3, а также любое положительное целое значение, большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение, равное 1.

Тип атрибута указывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которым относится рассматриваемый класс. Можно также определять тип атрибута в зависимости от языка программирования, который будет использоваться для реализации данной модели.

Например:

*имя\_студента [1..5] String здесь имя\_студента*

является именем атрибута*, тип атрибута String (строка)*

Исходное значение служит для задания некоторого начального значения для соответствующего атрибута в момент создания отдельного экземпляра класса.

Например:

*имя студента [1..5] : String = Иван.*

Строка-свойство служит для указания фиксированных значений атрибута. Эти значения не могут быть изменены в программе при работе с данным типом объектов. При отсутствии строки-свойства значение соответствующего атрибута может быть изменено в программе. Например, строка-свойство в записи атрибута стипендия Integer = {$50} означает фиксированную сумму стипендии для всех объектов класса "Студент". Запись данного атрибута в виде стипендия Integer = $50 означает, что при создании нового экземпляра Студент для него устанавливается по умолчанию стипендия в 50 долларов.

Операцией класса (методом класса) называется именованный сервис, который предоставляется по требованию любым объектом данного класса. Другими словами, операция – это абстракция того, что можно делать с объектом. Класс может содержать любое число операций (в частности, не содержать ни одной операции). Набор операций класса является общим для всех объектов данного класса.

Операции класса определяются в секции, расположенной ниже секции атрибутов. При этом можно ограничиться только указанием имен операций, оставив детальную спецификацию выполнения операций на более поздние этапы моделирования. Для именования операций рекомендуется использовать глаголы, соответствующие ожидаемому поведению объектов данного класса. Описание операции имеет следующий вид:

*<квантор видимости><имя операции> (список параметров)*

*<выражение типа возвращаемого значения>{строка-свойство}*

Квантор видимости принимает такие же значения, как и в случае атрибутов класса, и может быть опущен. Вместо условных графических обозначений также можно записывать соответствующее ключевое слово: *public* (общедоступный), *protected* (защищенный), *private* (закрытый).

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции, и поэтому должна быть уникальной в пределах данного класса. Имя является единственным обязательным элементом синтаксического обозначения операции.

Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем виде:

*<вид параметра><имя параметра>:<выражение типа>=*

*<значение параметра по умолчанию>.*

Вид параметра – это одно из ключевых слов *in* (в), *out* (из) или *inout* со значением *in* по умолчанию. Имя параметра есть идентификатор соответствующего формального параметра. Выражение типа зависит от конкретного языка программирования и описывает тип возвращаемого значения для соответствующего формального параметра.

Строка-свойство служит для определения значений свойств данного элемента. Строка-свойство может отсутствовать, если никакие свойства не специфицированы.

Операция, которая не может изменять состояние системы, обозначается строкой-свойством ({запрос, *query*}).

Описание операции на самом верхнем уровне объявляет эту операцию на весь класс, при этом данная операция наследуется всеми потомками данного класса. Если в некотором классе операция не выполняется, то такая операция может быть помечена как абстрактная (*abstract*). Можно также записать сигнатуру операции курсивом, чтобы обозначить ее абстрактной. Подчиненное появление записи данной операции без свойства {абстрактная} указывает на тот факт, что соответствующий класс-потомок может выполнять данную операцию в качестве своего метода.

Пример:

+создать() – обозначает абстрактную операцию по созданию отдельного объекта класса, которая является общедоступной (*public*) и не содержит формальных параметров. Эта операция не возвращает никакого значения после своего выполнения.

***2. Диаграмма пакетов (объектов)***показывает, из каких частей состоит проектируемая программная система, и как эти части связаны друг с другом.

Связь между пакетами фиксируют, если изменения в одном пакете могут повлечь за собой изменения в другом. Она определяется внешними связями классов и других ресурсов, объединенных в пакет. Возможны различные виды зависимости классов, например:

* объекты одного класса посылают сообщения объектам другого класса;
* объекты одного класса обращаются к компонентам объектов другого;
* объекты одного класса используют объекты другого в списке параметров классов и т. п.

Хорошим технологическим вариантом является тот, при котором каждый пакет включает интерфейс, содержащий описание всех ресурсов данного пакета, и взаимодействие пакетов осуществляется только через этот интерфейс. Изменения реализации ресурсов пакета в этом случае не затрагивает других пакетов. И только изменения в интерфейсе могут потребовать изменения пакетов, использующих ресурсы данного пакета.

Пакеты, с которыми связаны все пакеты программной системы, называют глобальными.

Интерфейсы таких пакетов необходимо проектировать особенно тщательно, так как изменения в них потребуют проверки всех пакетов разрабатываемой системы.

На рисунке 3.69 приведены обозначения нотации UML, которые допустимо использовать на диаграммах пакетов. Кроме указанных обозначений на диаграммах пакетов допустимо показывать обобщения (рисунок 3.70), что, как правило, подразумевает наличие единого интерфейса нескольких пакетов.

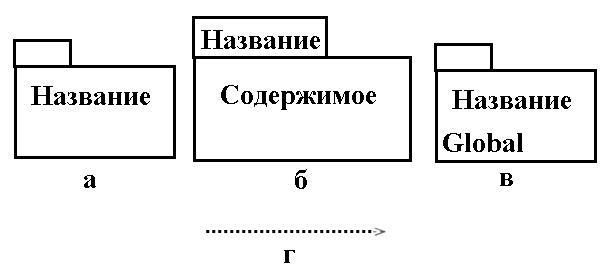


Рисунок 3.69 – Условные обозначения, применяемые на диаграммах пакетов: а – пакет; б – пакет с обозначением содержимого; в – глобальный пакет; г – зависимость классов

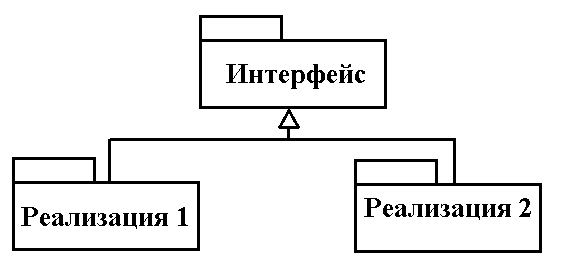


Рисунок 3.70 – Пример обозначения обобщения пакетов

В этом случае фиксируется связь от пакета-подтипа к пакету-супертипу.

На рисунке 3.71 приведен примердиаграммы пакетов системы решения комбинаторно-оптимизационных задач.

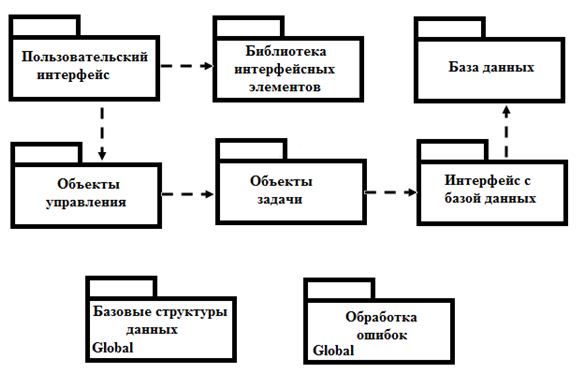


Рисунок 3.71 – Диаграмма пакетов системы решения комбинаторно-оптимизационных задач

Анализ концептуальной модели и вариантов использования позволяют выделить следующие группы классов или пакеты:

* пользовательский интерфейс – классы, реализующие объекты интерфейса с пользователем;
* библиотека интерфейсных компонентов – классы, реализующие интерфейсные компоненты: окна, кнопки, метки и т. п.;
* объекты управления – классы, реализующие сценарии вариантов использования;
* объекты задачи – классы, реализующие объекты предметной области системы;
* интерфейс базы данных – классы, реализующие интерфейс с базой данных;
* база данных;
* базовые структуры данных – классы, реализующие внутренние структуры данных, такие, как деревья, n-связные списки и т. п.;
* обработка ошибок – классы исключений, реализующие обработку нештатных ситуаций.

Последние два пакета являются глобальными, так как их элементы могут использовать классы всех пакетов.

После определения основных пакетов разрабатываемого программного обеспечения переходят к детальному проектированию классов, входящих в каждый пакет. Классы-кандидаты, которые предположительно должны войти в конкретный пакет, показывают на диаграмме классов этапа проектирования и уточняют отношения между объектами указанных классов.

После определения основных пакетов разрабатываемого программного обеспечения переходят к детальному проектированию классов, входящих в каждый пакет. Классы-кандидаты, которые предположительно должны войти в конкретный пакет, показывают на диаграмме классов этапа проектирования и уточняют отношения между объектами указанных классов.

Основой для проектирования классов является уточнение взаимодействия объектов этих классов в процессе реализации вариантов использования. При этом применяют диаграммы последовательностей и диаграммы кооперации.

***3. Диаграмма Use Case***(диаграмма прецедентов, вариантов использования) показывает набор элементов Use Case, актеров и их отношений. Эти диаграммы нужны при организации и моделировании поведения системы, задании требований заказчика к системе.

Разработку спецификаций программного обеспечения начинают с анализа требований к функциональности, указанных в техническом задании. В процессе анализа выявляют внешних пользователей разрабатываемого программного обеспечения и перечень отдельных аспектов его поведения в процессе взаимодействия с конкретными пользователями.

Прецеденты (варианты использования – Use Cases) – это подробные процедурные описания вариантов использования системы всеми заинтересованными лицами, а также внешними системами, т. е. всеми, кто (или что) может рассматриваться как актеры (actors) – действующие лица. По сути это алгоритмы работы с системой с точки зрения внешнего наблюдателя. Прецеденты являются основой функциональных требований к системе, позволяют описывать границы проектируемой системы, ее интерфейс, а затем выступают как основа для тестирования системы заказчиком с помощью приемочных тестов.

В зависимости от цели выполнения конкретной задачи различают следующие варианты использования:

* основные, обеспечивают выполнение функций проектируемой системы;
* вспомогательные, обеспечивают выполнение настроек системы и ее обслуживание;
* дополнительные, служат для удобства пользователя (реализуются в том случае, если не требуют серьезных затрат каких-либо ресурсов ни при разработке, ни при эксплуатации).

Пример.

Анализ функциональных требований и пользователей системы тестирования (модуль обучающей системы).

Система тестирования необходима следующим заинтересованным лицам:

* обучаемому (студенту);
* составителю тестов (преподавателю);
* преподавателю, принимающему экзамен;
* сотруднику деканата, осуществляющему контроль за успеваемостью;
* администратору сети и баз данных учебного учреждения.

На начальном этапе создания системы мы можем ограничиться только двумя важными для нас ролями действующих лиц:

* студент (тестируемый);
* администратор (он же преподаватель, он же составитель тестов).

Соответственно основные прецеденты (варианты использования) для нашей системы следующие:

Прецедент для студента:

* П1 – пройти тестирование.

Прецеденты для администратора:

* П2 – создать/изменить тест;
* П3 – просмотреть результаты тестирования;
* П4 – добавить/изменить пользователей и др.

Вариант использования можно описать кратко или подробно (таблицы 3.4, 3.5). Краткая форма описания содержит название варианта использования, его цель, действующих лиц, тип варианта использования (основной, второстепенный или дополнительный) и его краткое описание.

Таблица 3.4 – Краткое описание варианта использования (данный пример)

|  |  |
| --- | --- |
| Название варианта | Прохождение теста |
| Цель | Получение оценки |
| Действующие лица (актеры) | Студент |
| Краткое описание | Регистрация студента, запуск теста, выбор ответа или ввод ответа, завершение теста, получение оценки |
| Тип варианта | Основной |

Таблица 3.5 – Подробное описание варианта использования "Прохождение теста"

|  |  |
| --- | --- |
| Действия исполнителя | Отклик системы |
| 1. Студент вводит свои данные (ФИО, Группа), т.е. регистрируется в системе | 2. Система создает на диске файл с результатом тестирования и предлагает выбрать тест |
| 3. Студент выбирает тест | 4. Система запускает тест |
| 5. Студент последовательно отвечает на вопросы | 6. Система регистрирует правильные и неправильные ответы |
| 7. Студент завершает тестирование | 8. Система подсчитывает процент правильных ответов |
| 9. Студент ожидает результат | 10. Система демонстрирует результат и предлагает его сохранить |
| 11. Студент решает, сохранить результат или нет | 12. Если выбрано сохранение, система записывает его в файл |
| 13. Студент завершает работу | 14. Система завершает работу |

Для большей наглядности используют диаграммы вариантов использования.

На рисунке 3.72 приведены условные обозначения, которые применяют при изображении диаграмм прецедентов.

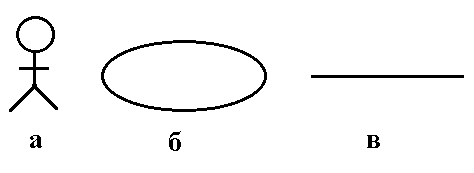


Рисунок 3.72 – Условные обозначения диаграмм прецедентов: а – актер; б – вариант использования; в – связь

Приведем диаграмму прецедентов для вышеописанного примера (рисунок 3.73).

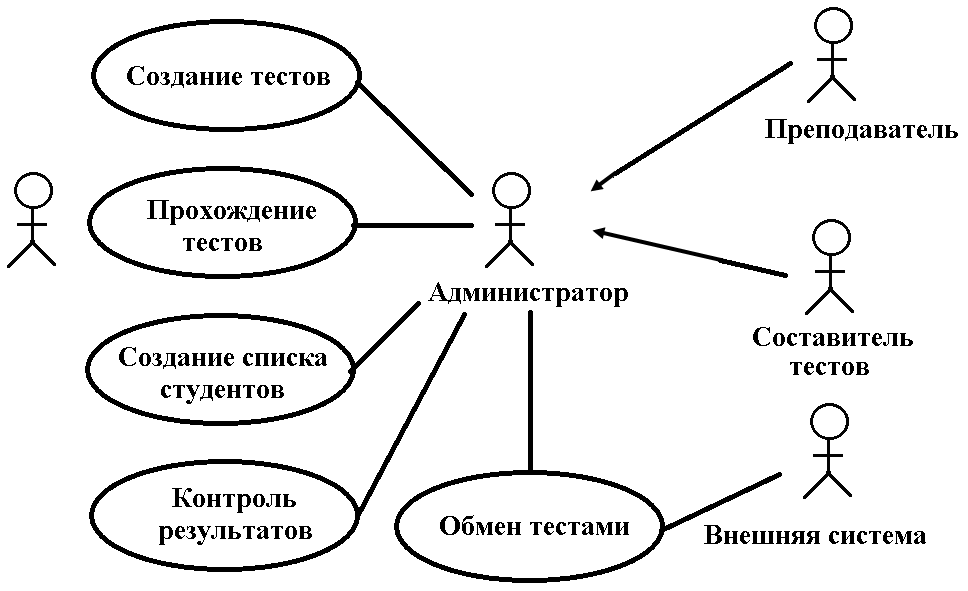


Рисунок 3.73 – Диаграмма вариантов использования тестовой системы

Все варианты использования определить, как правило, не удается, новые варианты фиксируют постоянно, даже в процессе эксплуатации. Но чем больше вариантов выявлено в процессе уточнения спецификаций, тем лучше, так как при этом получают более точную модель предметной области, что уменьшает вероятность ее пересмотра при добавлении функций.

***4. Диаграмма последовательности*** *–* это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени.

Диаграммы последовательности и диаграммы сотрудничества – это разновидности диаграмм взаимодействия.

Диаграмма последовательностей системы (sequence diagram) – графическая модель, которая для определенного сценария варианта использования показывает динамику взаимодействия объектов во времени.

Для построения диаграммы последовательностей системы необходимо:

* идентифицировать каждое действующее лицо (объект) и изобразить для него линию жизни. Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия (см. объект 1 на рисунке 3.74). Правее изображается другой объект, который непосредственно взаимодействует с первым, и т. д.;
* из описания варианта использования определить множество системных событий и их последовательность;
* изобразить системные события в виде линий со стрелкой на конце между линиями жизни действующих лиц и системы, а также указать имена событий и списки передаваемых значений.

На диаграмме последовательности изображаются только те объекты, которые непосредственно участвуют во взаимодействии, и не показываются возможные статические ассоциации с другими объектами. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни (рисунок 3.74). Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием. При этом вся запись подчеркивается, что является признаком объекта, который представляет собой экземпляр класса.

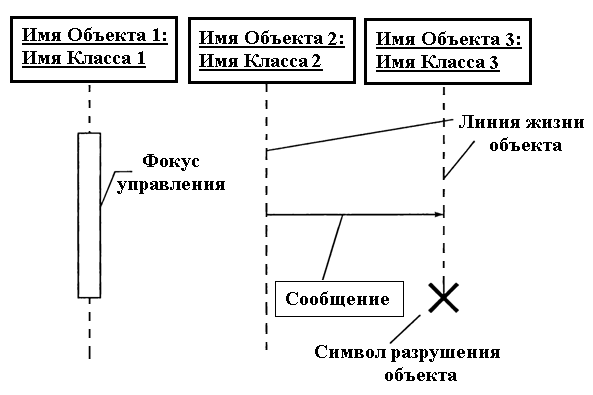


Рисунок 3.74 – Различные графические примитивы диаграммы последовательности

*Линия жизни* объекта (*object lifeline*) служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. На диаграмме линия жизни изображается пунктирной вертикальной линией, ассоциированной с единственным объектом. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней (объекты 1 и 2 на рисунке 3.74).

Если объекты разрушаются в какой-то момент для освобождения ресурсов системы, то их линия жизни обрывается в момент уничтожения. Для обозначения такого момента в языке UML используется специальный символ в форме латинской буквы "X" (объект 3 на рисунке 3.74). Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий.

Объекты на диаграмме последовательности могут находиться в двух состояниях: активном, непосредственно выполняя какие-либо действия, и пассивном, ожидая сообщения от других объектов. Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название *фокуса управления* (*focus of control*). Фокус управления изображается в форме вытянутого узкого прямоугольника (см. объект 1 на рисунке 3.74), верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а ее нижняя сторона – окончание фокуса управления (окончание активности).

Диаграмма последовательности описывает динамику взаимодействий между множеством объектов. Каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой. *Сообщение* (*message*) представляет собой законченный фрагмент информации, который отправляется одним объектом другому. Объект, принявший сообщение, должен отреагировать на него какой-либо последовательностью действий, направленных на решение поставленной задачи.

Графически сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности (рисунок 3.74).

На рисунке 3.75 можно увидеть пример построения диаграммы последовательности для моделирования процесса телефонного разговора с использованием обычной телефонной сети. Объектами в этом примере являются: два абонента А и Б, два телефонных аппарата, коммутатор и сам разговор как объект моделирования.

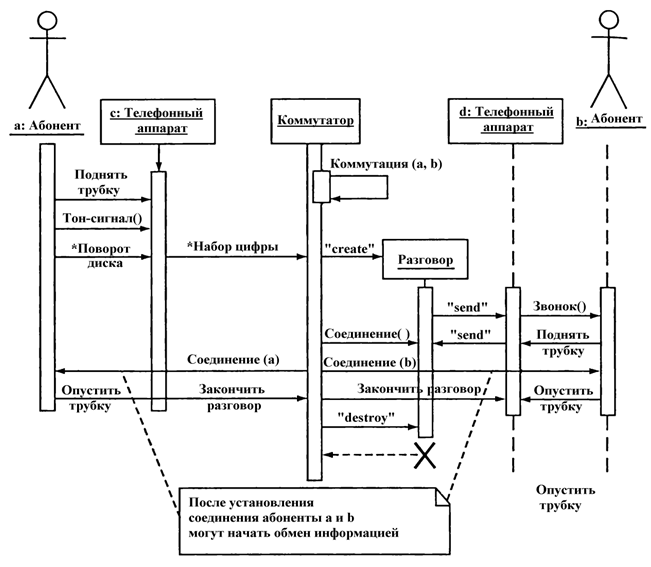


Рисунок 3.75 – Пример диаграммы последовательностей состояний

***5. Диаграмма сотрудничества***(диаграмма кооперации) – это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения.

Диаграмма кооперации является альтернативным диаграмме последовательностей способом представления взаимодействия объектов. Она показывает потоки данных между объектами классов, что позволяет уточнить отношения между ними.

На диаграмме изображаются участвующие во взаимодействии объекты (в виде прямоугольников, содержащих имя объекта, имя его класса и значения атрибутов, если они имеются), а также указываются ассоциации между этими объектами, если необходимо, указывают имя ассоциации и роли объектов в данной ассоциации. Дополнительно могут быть изображены динамические связи – потоки сообщений. Они представляются также в виде соединительных линий между объектами, над которыми располагается стрелка с указанием направления, имени сообщения и порядкового номера в общей последовательности инициализации сообщений. Номера служат для синхронизации сообщений, так как на диаграмме кооперации прямо не указывается время.

Пример.Разработать диаграмму коопераций, иллюстрирующую взаимоотношения компании с составляющими ее отделами и сотрудниками.

Компания организует и расформировывает отделы, принимает и увольняет сотрудников, выдает задание отделам. Сотрудники выполняют работу и сдают отчеты начальникам отделов.

Решение приведено на рисунке 3.76.

На приведенной диаграмме объекты классов "Отдел" и "Сотрудник" реализованы мульти объектами.



Рисунок 3.76 – Пример диаграммы кооперации

***6. Диаграмма схем состояний***показывает конечный автомат, представляет состояния, переходы, события и действия. Диаграммы схем состояний (*statechart diagram*) обеспечивают динамическое представление системы. Они особенно важны при моделировании поведения интерфейса, класса или сотрудничества. Эти диаграммы выделяют такое поведение объекта, которое управляется событиями, что особенно полезно при моделировании реактивных систем (системы взаимодействуют с окружающей средой посредством обмена сообщениями в темпе, задаваемом средой).

Главное предназначение этой диаграммы – описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение элемента модели в течение его жизненного цикла. Диаграммы состояний используются для моделирования динамических аспектов системы. По большей части под этим подразумевается моделирование поведения реактивных объектов. Реактивным называется объект, поведение которого лучше всего характеризуется его реакцией на события, произошедшие вне его собственного контекста. У реактивного объекта есть четко выраженный жизненный цикл, когда текущее поведение обусловлено его прошлым. Если внешние действия, изменяющие состояния системы, инициируются в произвольные случайные моменты времени, то говорят об асинхронном поведении модели.

Диаграмма состояний по существу является графом специального вида, который представляет некоторый *автомат*. Вершинами этого графа являются состояния, которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы более детального представления отдельных элементов модели.

Простейший пример автомата с двумя состояниями демонстрирует ситуацию с исправностью компьютера. Здесь рассматриваются два самых общих состояния: "исправен" и "неисправен" и два перехода: "выход из строя" и "ремонт". Графически эта информация может быть представлена в виде изображенной Диаграммы состояний компьютера (рисунок 3.77).

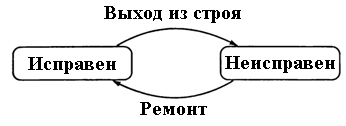


Рисунок 3.77 – Пример диаграммы состояний

Основными понятиями, описывающими автомат, являются состояние и переход. Предполагается, что система находится в каком-либо состоянии в течение некоторого времени, тогда как переход объекта из состояния в состояние происходит мгновенно.

*Состояние* (*state*) – это ситуация в жизни объекта, на протяжении которой он удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого-то события.

Состояние на диаграмме изображается прямоугольником со скругленными вершинами (рисунок 3.78), который может быть разделен горизонтальной линией на две секции.



Рисунок 3.78 – Графическое изображение состояний на диаграмме состояний

В прямоугольнике может располагаться "Имя состояния" (первая секция) и "Список внутренних действий в данном состоянии" (вторая секция). При этом под *действием* (*action*) в языке UML понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения.

Имя состояния – это строка текста, начинающаяся с заглавной буквы, которая раскрывает содержательный смысл данного состояния. Имя является необязательным элементом. Рекомендуется в качестве имени использовать глаголы в настоящем времени (звенит, печатает, ожидает) или соответствующие причастия (занят, свободен, передано, получено).

Список внутренних действий содержит перечень внутренних действий или деятельностей, которые выполняются в процессе нахождения моделируемого элемента в данном состоянии. Каждое из действий 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, которые принадлежат области имен или контексту моделируемого объекта. Если список выражений действия пустой, то разделитель в виде наклонной черты "/" может не указываться.

На рисунке 3.79 показан пример состояния: "Считывает запись после открытия файла, содержащего несколько записей".

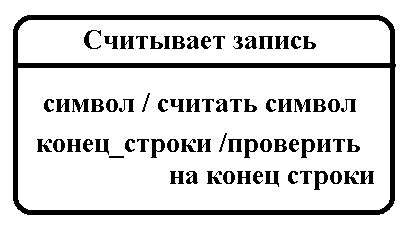


Рисунок 3.79 – Пример состояния с непустой секцией внутренних действий

Простой *переход* (*simple transition*) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим. Пребывание моделируемого объекта в первом состоянии может сопровождаться выполнением некоторых действий, а переход во второе состояние будет возможен после завершения этих действий, а также после удовлетворения некоторых дополнительных условий. В этом случае говорят, что переход срабатывает. До срабатывания перехода объект находится в исходном состоянии, или в источнике (не путать с начальным состоянием – это разные понятия), а после его срабатывания объект находится в последующем состоянии (целевом состоянии).

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая направлена в целевое состояние (например, "выход из строя" на рисунке 3.77). Рядом с линией может находиться строка текста, описывающая событие-триггер, вызывающий переход (в этом случае переход будет триггерным), и сторожевое условие, по которому осуществляется переход. В примере на рисунке 3.77 переход сработает при возникновении события – "выход из строя".

*Событие* (*еvent*) – это спецификация существенного факта, который происходит во времени и пространстве. В контексте автоматов событие – это стимул, способный вызвать срабатывание перехода.

*Сторожевое условие* (*guard condition*), если оно есть, всегда записывается в прямых скобках после события-триггера и представляет собой некоторое булевское выражение (выражение, результатом которого является "истина" или "ложь" ).

Пример диаграммы состояний почтовой программы-клиента показан на рисунке 3.80.

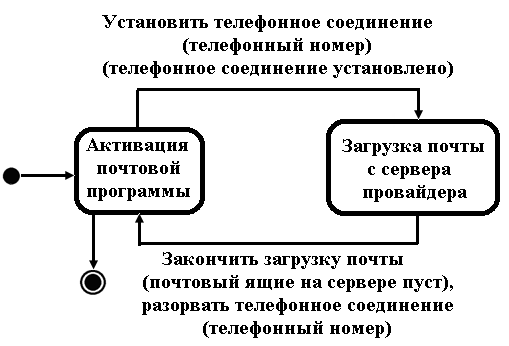


Рисунок 3.80 –Диаграмма состояний для моделирования почтовой программы-клиента

***7. Диаграмма деятельности* (*activity diagram*)**  *–* специальная разновидность диаграммы схем состояний, которая показывает поток от действия к действию внутри системы. Диаграммы деятельности обеспечивают динамическое представление системы. На этапе анализа требований и уточнения спецификаций диаграммы деятельностей позволяют конкретизировать основные функции разрабатываемого программного обеспечения.

Под деятельностью понимают задачу (операцию), которую необходимо выполнить вручную или с помощью средств автоматизации. Каждому варианту использования соответствует своя последовательность задач. В теоретическом плане диаграммы деятельности являются обобщенным представлением алгоритма, реализующего анализируемый вариант использования. Графически состояние действия представляется прямоугольником со скругленными углами (рисунок 3.81). Внутри этой фигуры записывается выражение действия (*action-expression*), которое должно быть уникальным в пределах одной диаграммы деятельности.

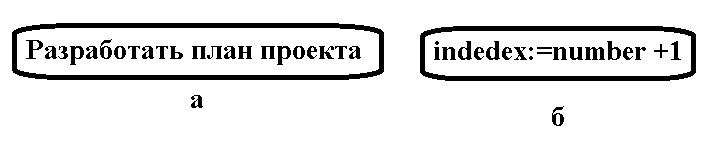


Рисунок 3.81 – Графическое изображение состояния действия: а – простое действие; б – выражение

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами (рисунок 3.81, а). Если это возможно, то допускается запись действия на том языке программирования, на котором предполагается реализовывать конкретный проект (рисунок 3.81, б).

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояния (рисунок 3.82). Диаграмму деятельности принято располагать таким образом, чтобы действия следовали сверху вниз. В этом случае начальное состояние будет изображаться в верхней части диаграммы, а конечное – в ее нижней части.

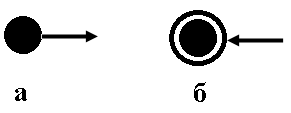


Рисунок 3.82 – Графическое изображение: а – начальное состояние; б – конечное состояния

При построении диаграммы деятельности используются только нетриггерные переходы, т. е. такие, которые срабатывают сразу после завершения деятельности или выполнения соответствующего действия. Этот переход переводит деятельность в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой.

Если из состояния действия выходит единственный переход, то он может быть не помечен. Если последовательно выполняемая деятельность должна разделиться на альтернативные ветви в зависимости от значения некоторого промежуточного результата (такая ситуация получила название ветвления, а для ее обозначения применяется специальный символ), то таких переходов несколько и сработать может только один из них. Именно в этом случае для каждого из таких переходов должно быть явно записано условие в квадратных скобках. При этом для всех выходящих из некоторого состояния переходов должно выполняться требование истинности только одного из них.

Графически ветвление на диаграмме деятельности обозначается небольшим ромбом, внутри которого нет никакого текста (рисунок 3.83). В этот ромб может входить только одна стрелка от того состояния действия, после выполнения которого поток управления должен быть продолжен по одной из взаимно исключающих ветвей. Принято входящую стрелку присоединять к верхней или левой вершине символа ветвления. Выходящих стрелок может быть две или более, но для каждой из них явно указывается соответствующее условие перехода в форме булевского выражения.

На рисунке 3.83 представлен фрагмент известного алгоритма нахождения корней квадратного уравнения.

В общем случае после приведения уравнения второй степени к каноническому виду *а∙х∙х + b∙x+ с = 0* в случае отрицательного дискриминанта уравнение не имеет решения на множестве действительных чисел, и дальнейшие вычисления должны быть прекращены. При неотрицательном дискриминанте уравнение имеет решение, корни которого могут быть получены на основе конкретной расчетной формулы.

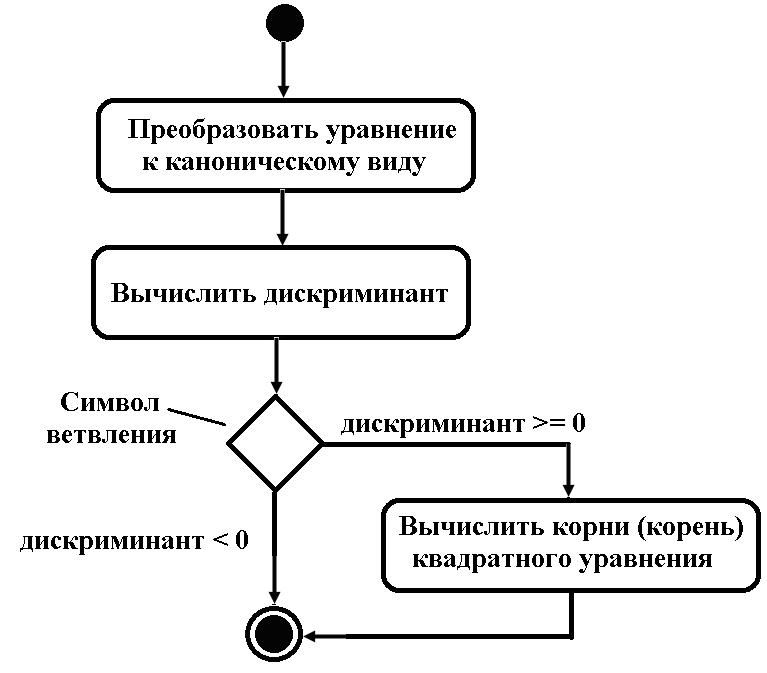


Рисунок 3.83 – Фрагмент диаграммы деятельности для алгоритма нахождения корней квадратного уравнения

Для представления параллельных процессов в языке UML используется специальный символ разделения и слияния параллельных вычислений или потоков управления. Таким символом является прямая черточка, которая изображается отрезком горизонтальной линии, толще чем основные сплошные линии диаграммы деятельности. При этом разделение (*concurrent fork*) имеет один входящий переход и несколько выходящих (рисунок 3.84, а). Слияние (*concurrent join*), наоборот, имеет несколько входящих переходов и один выходящий (рисунок 3.84, б).

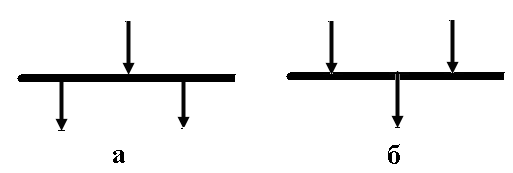


Рисунок 3.84 – Графическое изображение: а – разделения; б – слияния параллельных потоков управления

На рисунке 3.85 показан пример диаграммы деятельности процесса приготовления напитка.

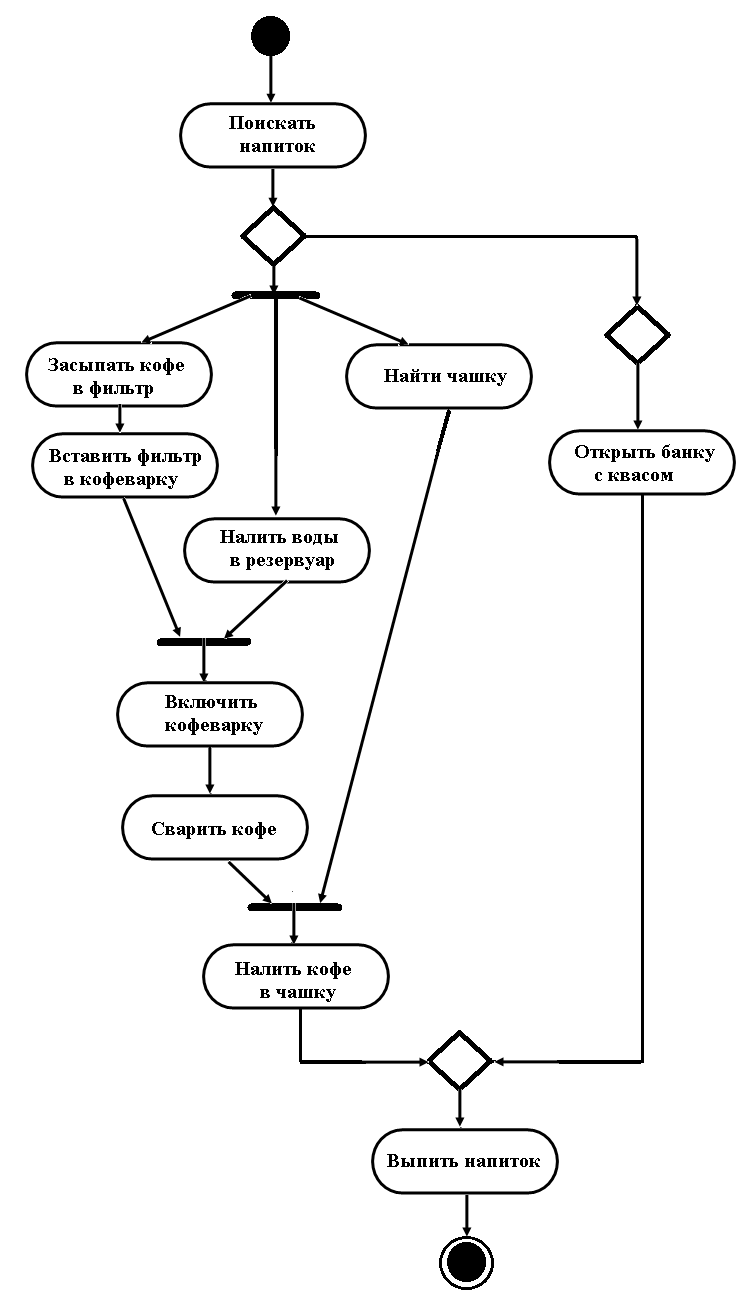


Рисунок 3.85 – Пример диаграммы деятельности

***8. Компонентная диаграмма***показывает организацию набора компонентов и зависимости между компонентами. Компонентные диаграммы обеспечивают статическое представление реализации системы. Они связаны с диаграммами классов и в компоненте обычно отображается один или несколько классов, интерфейсов или коопераций. В качестве физических компонент могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п.

Диаграммы компонентовприменяют при проектировании физической структуры разрабатываемо программного обеспечения. Эти диаграммы показывают, как выглядит программное обеспечение на физическом уровне, т. е. из каких частей оно состоит и как эти части связаны между собой.

Диаграммы компонентов оперируют понятиями компонент и зависимость. Под компонентамипонимают физические заменяемые части программного обеспечения, которые соответствуют некоторому набору интерфейсов и обеспечивают их реализацию. По сути дела, это отдельные файлы различных типов: исполняемые (.ехе), текстовые, графические, таблицы баз данных и т. п., составляющие разрабатываемое программное обеспечение. Условные графические обозначения компонентов различных типов приведены на рисунке 3.86.

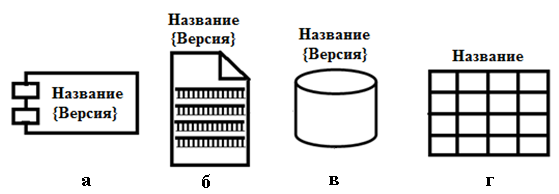


Рисунок 3.86 – Условные обозначения компонентов в UML: а – программный компонент; б – файл; в – база данных; г – таблица БД

Зависимость между компонентами фиксируют, если один компонент содержит некоторый ресурс (модуль, объект, класс и т. д.), а другой – его использует. Качество компоновки оценивают по количеству и типу связей между компонентами, т. е. по степени независимости компонентов.

На диаграмме компонентов зависимость обозначают пунктиром со стрелкой на конце. На рисунке 3.87 в качестве примера приведена диаграмма компонентов системы решения комбинаторно-оптимизационных задач с интерфейсами через которые компоненты связаны между собой.

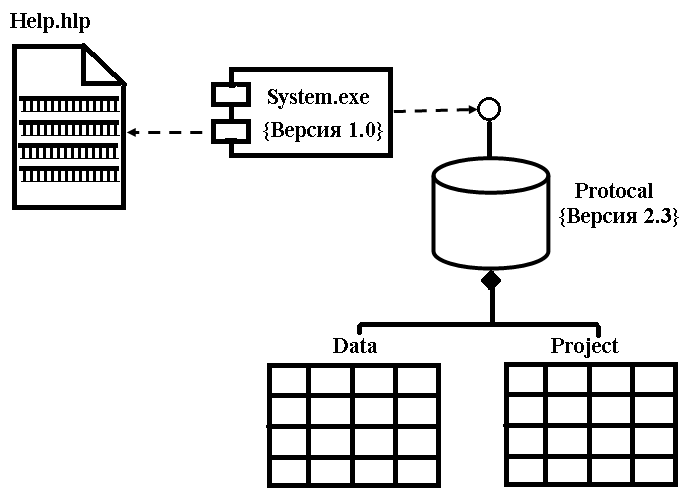


Рисунок 3.87 – Диаграмма компонентов системы решения комбинаторно-оптимизационных задач с указанием интерфейса

Кроме этого, на диаграмме компонентов допустимо уточнять зависимость между компонентами, используя обозначения обобщения, ассоциации, композиции, агрегирования и реализации. На рисунке 3.87 показано, что база данных включает (отношение композиции) две таблицы.

***9. Диаграмма размещения***(диаграмма развертывания)

При физическом проектировании распределенных программных систем необходимо определить наиболее оптимальный вариант размещения программных компонентов на реальном оборудовании в локальной или глобальной сетях. Диаграмма размещенияотражает физические взаимосвязи между программными и аппаратными компонентами системы. Каждой части аппаратных средств системы, например, компьютеру или датчику, на диаграмме размещения соответствует узел. *Соединения узлов* означают наличие в системе соответствующих коммуникационных каналов. Внутри узлов указывают размещенные на данном оборудовании программные компоненты разрабатываемой программной системы, сохраняя указанные на диаграмме компонентов отношения зависимости.

С точки зрения диаграммы размещения локальная и глобальная сети – это тоже узлы, которые обладают некоторой спецификой. На рисунке 3.88 показаны условные обозначения узлов (процессора и устройства) на диаграмме размещения.

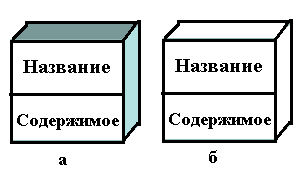


Рисунок 3.88 – Условные обозначения диаграммы размещения: а –процессор (компьютер); б –устройство

Пример.Диаграмма размещения для системы учета успеваемости студентов.

Локальная сеть деканата связывает сервер деканата и компьютеры декана, его заместителей и сотрудников деканата, отвечающих за занесение информации в базу данных. Серверная часть системы и база данных помещена на сервере деканата. На компьютерах локальной сети функционируют соответствующие клиентские части приложения (рисунок 3.89).

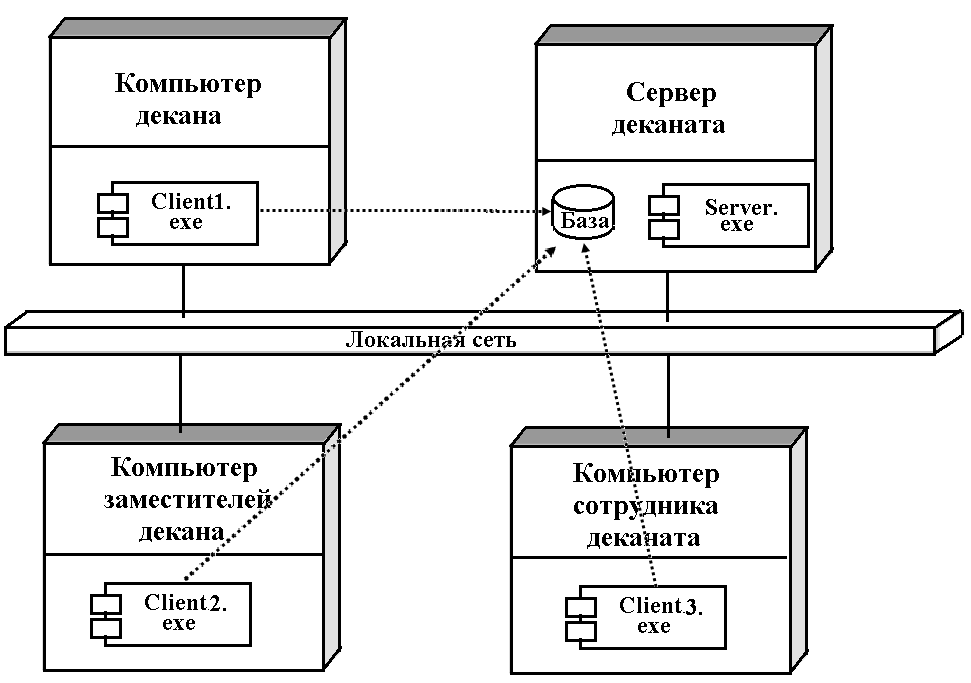


Рисунок 3.89 – Диаграмма размещения в локальной сети системы учета успеваемости студентов

## Механизмы расширения в UML

UML – развитый язык, имеющий большие возможности, но даже он не может отразить все нюансы, которые могут возникнуть при создании различных моделей. Поэтому UML создавался как открытый язык, допускающий контролируемые расширения. Механизмами расширения в UML являются:

* ограничения;
* теговые величины;
* стереотипы.

*Ограничение* (*constraint*) расширяет семантику строительного UML-блока, позволяя добавить новые правила или модифицировать существующие. Ограничение показывают как текстовую строку, заключенную в фигурные скобки {}. Например, на рисунке 3.90 введено простое ограничение на свойство *сумма* класса *Сессия Банкомата –* его значение должно быть кратно 20. Кроме того, здесь показано ограничение на два элемента (две ассоциации), оно располагается возле пунктирной линии, соединяющей элементы, и имеет следующий смысл – владельцем конкретного счета не может быть и организация, и физическое лицо.

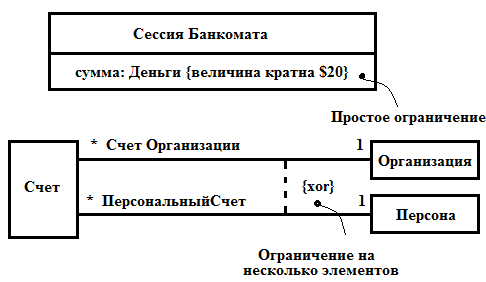


Рисунок 3.90 –Ограничения

*Теговая величина* (*tagged value*) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного элемента. Теговую величину показывают как строку в фигурных скобках {}. Строка имеет вид

*имя теговой величины = значение*

Иногда (в случае предопределенных тегов) указывается только имя теговой величины.

При работе с продуктом, имеющим много реализаций, полезно отслеживать версию и автора определенных блоков. Версия и автор не принадлежат к основным понятиям UML. Они могут быть добавлены к любому строительному блоку (например, к классу) введением в блок новых теговых величин. Например, на рисунке 3.91 класс ТекстовыйПроцессор расширен путем явного указания его версии и автора.

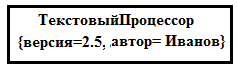


Рисунок 3.91 –Расширение класса

*Стереотип* (stereotype) расширяет словарь языка, позволяет создавать новые виды строительных блоков, производные от существующих и учитывающие специфику новой проблемы. Элемент со стереотипом является вариацией существующего элемента, имеющей такую же форму, но отличающуюся по сути. У него могут быть дополнительные ограничения и теговые величины, а также другое визуальное представление. Он иначе обрабатывается при генерации программного кода. Отображают стереотип как имя, указываемое в двойных угловых скобках (или в угловых кавычках).

Примеры элементов со стереотипами приведены на рисунке 3.92. Стереотип "*exception*" говорит о том, что класс ПотеряЗначимости теперь рассматривается как специальный класс, которому разрешается только генерация и обработка сигналов исключений. Особые возможности метакласса получил класс ЭлементМодели. Здесь показано применение стереотипа "call" к отношению зависимости (у него появился новый смысл).

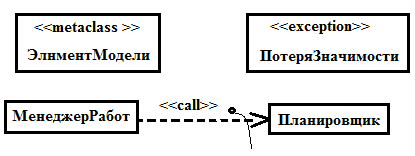


Рисунок 3.92 –Стереотипы

Таким образом, механизмы расширения позволяют адаптировать UML под нужды конкретных проектов и под новые программные технологии. Возможно добавление новых строительных блоков, модификация спецификаций существующих блоков и даже изменение их семантики. Конечно, очень важно обеспечить контролируемое введение расширений.

**3.8 Контрольные вопросы**

3.1 Дайте характеристику технологии программирования и архитектуре ПО в 50-60-е годы прошлого столетия.

3.2 Дайте характеристику технологии программирования и архитектуре ПО в 70-80-е годы прошлого столетия.

3.3 Дайте характеристику технологии программирования и архитектуре ПО от 90-х годов прошлого столетия до нашего времени.

3.4 Что такое компонентный подход и CASE-технологии?

3.5 Какие процессы составляют основы проектирования программных систем?

3.6 Перечислите основные характеристики программного модуля. Чем характеризуется размер модуля? Что такое рутинность модуля?

3.7 Что такое связность модуля?

3.8 Что такое сцепление модулей?

3.9 Как строится модульная структура ПС при восходящем проектировании?

3.10 Как строится модульная структура ПС при нисходящем проектировании?

3.11 Какие методы проектирования ПО при нисходящем подходе?

## 3.12 Чем характеризуется иерархическая структура программной системы?

3.13 Перечислите основные принципы структурного подхода разработки ПО.

3.14 Как осуществляется анализ требований и определение спецификаций при структурном подходе ( диаграммы DFD, ERD, STD, SADT)?

3.15 Какие вспомогательные средства проектирования ПО используются при структурном подходе (функциональная схема, схемы Джексона)?

3.16 Как осуществляется проектирование ПО методом пошаговой детализации или уточнения?

3.17 Перечислите основные принципы объектно-ориентированного представления программных систем (абстрагирование, инкапсуляция, модульность, иерархическая организация).

### 3.18 Дайте общую характеристику объектов.

### 3.19 Какие виды отношений могут быть между объектами?

### 3.20 Что такое класс? Какие виды отношений могут быть между классами?

3.21 Что такое архитектура ПО?

### 3.22 Как осуществляется анализ области решений?

3.23 Как разрабатывается и оценивается архитектура ПО на основе сценариев?

3.24 Что такое UML?

## 3.25 Какие виды отношений в UML?

3.26 Охарактеризуйте диаграмму классов.

3.27 Охарактеризуйте диаграмму объектов (пакетов).

3.28 Охарактеризуйте диаграмму Use Case (диаграмма прецедентов или вариантов использования).

3.29 Охарактеризуйте диаграмму последовательности.

3.30 Охарактеризуйте диаграмму сотрудничества (кооперации).

3.31 Охарактеризуйте диаграмму UML: диаграмма состояний.

3.32 Охарактеризуйте диаграмму деятельности.

3.33 Охарактеризуйте диаграмму компонентов.

3.34 Охарактеризуйте диаграмму размещения (развертывания).

## 3.35 Назовите механизмы расширения в UML.