## Тема 3.3. Теоретические предпосылки объектно-ориентированного проектирования и принципы моделирования программного обеспечения

**3.3.1. Основные предпосылки и идеи объектно-ориентированного**

**программирования**

**3.3.2. Объектно-ориентированная декомпозиция и объектная модель**

**3.3.3. Основные принципы моделирования программного обеспечения**

### 3.3.1.Основные предпосылки и идеи объектно-ориентированного программирования

Как мы видим окружающий мир? Ответ зависит от нашего прошлого. Ученый может видеть мир как множество молекулярных структур. Художник видит мир как набор форм и красок. А кто-то может сказать, что мир – это множество ***вещей***. Вероятно, первая мысль, которая пришла в голову, была: – какое значение имеет то, как кто-то видит мир? Однако это имеет большое значение для программиста, которому необходимо написать программу, **моделирующую реальный мир.**

Все мы видим мир как составляющие его вещи. Дом – вещь. Имущество, которое находится в доме, – вещи. То, что выкидываете, – вещи, покупаете вещи. Вещи, которые мы имеем, например дом, сделаны из других вещей, таких как окна или двери.

Технически говоря, любая вещь – это объект. То есть, дом – объект. Вещи, которые находятся в доме, – объекты. Вещи, которые вы выбрасываете, – объекты, и вещи, которые вы хотите ку­пить, – тоже объекты. Все мы, независимо от нашего прошлого, видим окружающий мир как множество объектов. Объект – это: **человек**, **место**, **вещь**, **понятие, процесс** и, возможно, **событие.**

Если проанализировать, каким образом человек решает разнообразные практические задачи в окружающем его мире, то можно понять, что этот мир также является объектно-ориентированным. Например, чтобы попасть на работу, человек, как правило, взаимодействует с таким объектом, как транспортное средство. Транспортное средство, в свою очередь, состоит из объектов, которые, взаимодействуя друг с другом, приводят его в движение, благодаря чему чело­век реализует свою задачу – добирается до нужного пункта. При этом ни водитель, ни пассажир не обязаны знать, каким образом взаимодействуют объекты, из которых состоит транспортное средст­во.

В объектно-ориентированной технологии, как и в реальном мире, пользователи про­грамм изолированы от логической схемы, необходимой для выполнения задач. Например, для пе­чати страницы в текстовом редакторе пользователь вызывает определенную функцию нажатием кнопки на панели инструментов, но не видит происходящих при этом внутренних процессов. При печати страницы во время работы программы происходит сложное взаимодействие объектов, ко­торые, в свою очередь, взаимодействуют с принтером.

Сделаем несколько определений.

***Объектно-ориентированное программирование (ООП)*** – это стиль программирования, кото­рый фиксирует поведение реального мира таким способом, при котором детали его реализации скрыты. Применение такой технологии позволяет представить структуру программы в виде мно­жества взаимодействующих друг с другом ***объектов*.** В результате взаимодействия, осуще­ствляемого путем передачи сообщений между объектами, реализуются заданные функции про­граммы. Приняв сообщение, объект может выполнить определенное действие, называемое ***мето­дом*.**

***Объектно-ориентировання технология (ООТ)*** в настоящий момент является бурно развивающейся и одной из самых перспективных технологий программирования.

Еще раз обратим внимание на то, что человек привык существовать в мире, который состоит из объектов. ***Объект*** – это понятие, которое используется интуитивно с самого момента рождения, и, следовательно, попытки перенести работу с объектами в достаточно абстрактную по своей сути область, какой и является разработка программного обеспечения, естественны.

С другой стороны **ООТ** – это иной способ осмысления, формулирования и решения задач по созданию **ПО**. У программирования на объектно-ориентированном языке масса преимуществ, причем это относится не только к созданию более эффективного программного кода, но и к более простой модификации и расширению возможностей уже имеющихся программных систем.

***Объектно-ориентированные методы (ООМ) –*** понятие, которое является очень обширным, как, собственно, и ***объектное-ориентированное проектирование (ООПр), объектное-ориентированный анализ (ООА)*** и ***объектно-ориентированная технология (ООТ)*** и означает фактически целую философию разработки систем и представления знаний на базе мощного ***объектного*** подхода.

Исторически, развитие этой области началось с объектно-ориентированного программирования, и только совсем недавно появился большой интерес к другим вопросам.

Основные принципы ООП были сформулированы в 60-х годах при разработке языка **Simula**, дальнейшее их развитие пришлось на 70-е годы, когда был создан язык **Smalltalk**. В 80-е годы ин­терес к этим технологиям повысился и особую популярность завоевал такой язык как **С++.** ООП продолжало развиваться и в 90-е годы. В это время появился язык **Java**.

В 2002 году с выпуском Visual Studio .NET компания Microsoft представила на рынке компьютерных технологий новые языки ООП – C# и Visual Basic .NET**,** которые являются полностью объектно-ориентированными.

***Рассмотрим основные идеи и предпосылки развития объектно-ориентированного программирования.***

Оглядываясь на историю развития программирования, нельзя не заметить две сменяющих друг друга тенденции:

* смещение акцентов от программирования отдельных деталей к программированию более крупных компонент;
* развитие и совершенствование языков программирования высокого уровня.

Как известно (***Тема 3.1***), большинство современных программных систем «больше» и существенно «сложнее», чем были их предшественники даже несколько лет тому назад. Этот рост сложности вызвал большое число прикладных исследований по методологии проектирования, особенно, по ***декомпозиции, абстрагированию*** и ***иерархиям***. Создание более выразительных языков программирования пополнило достижения в этой области. ***Возникла тенденция перехода от языков, указывающих компьютеру, что делать (императивные языки), к языкам, описывающим ключевые абстракции проблемной предметной области (декларативные языки).***

Кроме того, в каждом следующем поколении менялись поддерживаемые языками механизмы абстракции.

Так, языки **первого поколения** ориентировались на научно-инженерные применения, и словарь этой предметной области был почти исключительно математическим. Такие языки, как **Fortran**, были созданы для упрощения программирования математических формул, чтобы освободить программиста от трудностей ассемблера и машинного кода. Первое поколение языков высокого уровня было шагом, приближающим программирование к предметной области и удаляющим от конкретной машины.

**Во втором поколении** языков основной тенденцией стало развитие алгоритмических абстракций. В это время мощность компьютеров быстро росла, а компьютерная индустрия позволила расширить области их применения. Главной задачей стало инструктировать машину, что делать: сначала прочти эти анкеты сотрудников, затем отсортируй их и выведи результаты на печать. Это было еще одним шагом к предметной области и удалением от конкретной вычислительной машины. В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появилась возможность решать все более сложные задачи, но это требовало умения обрабатывать самые разнообразные типы данных. Такие языки как **Algol-68** и затем **Pascal** стали поддерживать абстракцию данных. Программисты смогли описывать свои собственные типы данных. Это стало еще одним шагом к предметной области и удалением от привязки к конкретной машине.

В 70-е годы неадекватность более ранних языков программирования по написанию крупных программных систем стала очевидной, поэтому созданные новые языки имели механизмы, устраняющие это ограничение. Лишь немногие из этих языков смогли выжить, однако многие их принципы нашли отражение в новых версиях более ранних языков. Таким образом, мы получили языки **Smalltalk, Ada, C, C++.**

Наибольший интерес для дальнейшего изложения представляет класс языков, называемых ***объектными*** и ***объектно-ориентированными****,* которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции программного обеспечения и в первую очередь полностью объектно-ориентированные языки программирования **VB** и **C#**.

Для пояснения вышесказанного можно рассмотреть топологии (основные элементы языков программирования и их взаимодействие), характерные для каждого поколения.

На рис. 3.3-1 показана, типичная топология для большинства языков **первого поколения и первой стадии второго поколения**.

Для этих языков (например, **Fortran** и **Cobol**), основным строительными блоками являлись процедуры (процедурная технология). Программы, реализованные на таких языках, имели относительно простую структуру, состоящую только из глобальных данных и процедур. Стрелками на рисунке обозначено влияние подпрограмм на данные. В процессе разработки можно логически разделить разнотипные **Данные**, но механизмы языков практически не поддерживают такого разделения. Ошибка в какой-либо части программы может иметь далеко идущие последствия, так как область данных открыта всем подпрограммам. В больших программных системах трудно гарантировать целостность данных при внесении изменений в какую-либо часть системы. В процессе эксплуатации уже через короткое время возникала путаница из-за большого количества перекрестных связей между подпрограммами, запутанных схем управления, неясного смысла данных, что угрожало надежности системы и снижало ясность программы.

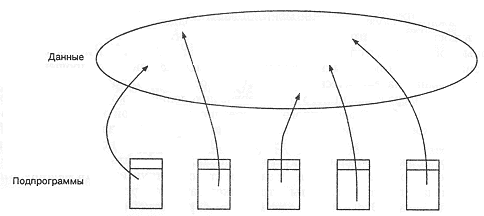


Рис. 3.3-1. Топология языков первого и начала второго поколения

**Топология языков позднего второго и раннего третьего поколения.** Начиная с середины 60-х годов стали осознавать роль процедур как важного промежуточного звена между решаемой задачей и компьютером. Процедуры возникли до 1950 года, но тогда они не были оценены в качестве абстракции. Их рассматривали как средства, упрощающие разработки программ. Но очень скоро стало ясно, что процедуры это абстрактные программные функции.

Использование процедур как механизма абстрагирования имело три существенных последствия. Во-первых, были разработаны языки, поддерживавшие разнообразные механизмы передачи параметров. Во-вторых, были заложен фундамент структурного программирования, что выразилось в языковой поддержке механизмов вложенности процедур и в научном исследовании структур управления и областей видимости. В-третьих, возникли методы структурного проектирования, стимулирующие разработчиков создавать большие системы, используя процедуры как готовые строительные блоки. Архитектура языков программирования этого периода (рис. 3.3-2), представляла собой вариации на темы предыдущего поколения. В нее были внесены кое-какие усовершенствования, в частности, усилено управление алгоритмическими абстракциями, но оставалась нерешенной проблема программирования «в большом» и проектирования данных.

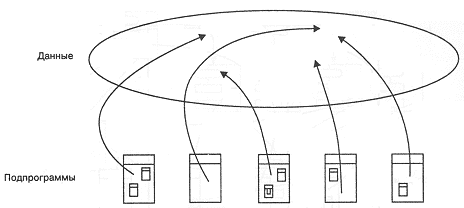


Рис. 3.3-2. Топология языков позднего второго и раннего третьего поколения

Начиная с **Fortran II** и далее (**языков конца третьего поколения)**, для решения задач программирования «в большом» начал развиваться новый важный механизм структурирования. Разрастание программных проектов означало увеличение размеров и коллективов программистов, а, следовательно, необходимость независимой разработки отдельных частей проекта. Ответом на эту потребность стал отдельно ***компилируемый модуль***, который сначала был просто более или менее случайным набором данных и процедур (рис. 3.1-3).

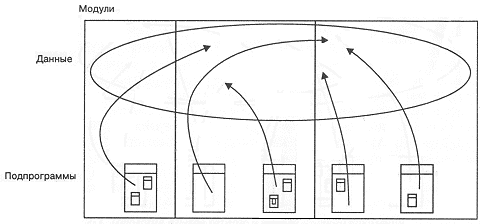


Рис.3.1-3*.* Топология языков конца третьего поколения

В такие модули собирали процедуры, которые, как казалось, скорее всего, будут изменяться совместно, и мало кто рассматривал их как новую технику абстракции. В большинстве языков этого поколения, хотя и поддерживалось модульное программирование, но не вводилось никаких правил, обеспечивающих согласование интерфейсов модулей. Программист, сочиняющий подпрограмму в одном из модулей, мог, например, ожидать, что ее будут вызывать с тремя параметрами: действительным числом, массивом из десяти элементов и целым числом, обозначающим логическое значение. Но в каком-то другом модуле, вопреки предположениям автора, эта подпрограмма могла по ошибке вызываться с фактическими параметрами в виде: целого числа, массива из пяти элементов и отрицательного числа. Аналогично, один из модулей мог завести общую область данных и считать, что это его собственная область, а другой модуль мог нарушить это предположение, свободно манипулируя с этими данными. К сожалению, поскольку большинство языков предоставляло в лучшем случае поддержку для работы с событиями и сообщениями абстрактных данных и типов, такие ошибки выявлялись только при выполнении программы.

**Топология объектных и объектно-ориентированных языков.** Значение абстрактных типов данных в разрешении проблемы сложности систем хорошо выраженов]: «Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не годится для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуациях сложность объектов, с которыми нужно работать, составляет основную часть сложности всей задачи». Осознание этого влечет два важных вывода.

Во-первых, возникают методы проектирования на основе ***потоков данных***, которые вносят упорядоченность в абстракцию данных в языках, ориентированных на алгоритмы. Во-вторых, появляется ***теория типов***, которая была воплощена в таком языке программирования, как Pascal.

Естественным завершением реализации этих идей, начавшейся с языка Simula и развитой в последующих языках в 1970-1980-е годы, стало появление таких языков, как Object Pascal, C++ и наконец, C#. На рис. 3.1-4 приведена топология таких языков применительно к задачам малой и средней степени сложности. Основным элементом конструкции в указанных языках служит ***модуль****,* составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.

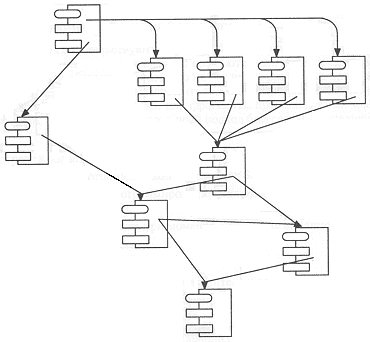


Рис. .3.1-4. Топология малых и средних приложений в объектных и объектно-

ориентированных языках

Другими словами: «Если процедуры и функции – глаголы, а данные – существительные, то процедурные программы строятся из глаголов, а объектно-ориентированные – из существительных». По этой же причине структура программ малой и средней сложности при объектно-ориентированном подходе представляется графом, а не деревом, как в случае алгоритмических языков. Кроме того, уменьшена или отсутствует область глобальных данных. Данные и действия организуются теперь таким образом, что основными логическими строительными блоками наших систем становятся ***классы и объекты, а не алгоритмы***.

В настоящее время мы продвинулись много дальше программирования   
«в большом» и предстали перед программированием «в огромном». Для очень сложных систем классы, объекты и модули являются необходимыми, но не достаточными средствами абстракции. К счастью, объектный подход масштабируется и может быть применен на все более высоких уровнях. Кластеры абстракций в больших системах могут представляться в виде многослойной структуры. На каждом уровне можно выделить группы объектов, тесно взаимодействующих для решения задачи более высокого уровня абстракции. Внутри каждого кластера мы неизбежно найдем такое же множество взаимодействующих абстракций. Это соответствует подходу к сложным системам.

Методы структурного проектирования помогают упростить процесс разработки сложных систем за счет использования базовых алгоритмов (процедур) как готовых строительных блоков. Аналогично, методы объектно-ориентированного проектирования созданы, чтобы помочь разработчикам применять мощные выразительные средства объектного и объектно-ориентированного программирования, использующего в качестве блоков классы и объекты.

Но в объектной модели отражается и множество других факторов – объектный подход зарекомендовал себя как унифицирующая идея всей компьютерной науки, применимая не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитектуры компьютерных систем. Причина такой широты в том, что ориентация на объекты позволяет нам справляться со сложностью систем самой разной природы.

  Объектно-ориентированный анализ и проектирование отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта. К сожалению, большинство программистов формально и неформально нацелено на применение только методов структурного проектирования. Разумеется, многие хорошие проектировщики создали и продолжают совершенствовать большое количество программных систем на основе этой методологии. Однако алгоритмическая декомпозиция помогает только до определенного предела, и обращение к объектно-ориентированной декомпозиции необходимо. Более того, при попытках использовать такие языки, как **C++** или **C#**, в качестве традиционных, алгоритмически ориентированных, мы не только теряем их внутренний потенциал –скорее всего результат будет даже хуже, чем при использовании обычных языков**С** и **Pascal**. Дать электродрель плотнику, который не слышал об электричестве, значит использовать ее в качестве молотка. Он согнет несколько гвоздей и разобьет себе пальцы, потому что электродрель мало пригодна для замены молотка.

Можно согласиться, что понятие объекта является центральным во всем, что относится к объектно-ориентированной методологии. Ранее мы определили объект как осязаемую ***сущность***, которая четко проявляет свое поведение. Другие определяют объекты как «***сущности, объединяющие процедуры и данные***, так как они производят вычисления и сохраняют свое локальное состояние». Определение объекта как сущности в какой-то мере отвечает на вопрос, но все же главным в понятии объекта является объединение идей ***абстракции данных и алгоритмов***: «В объектном подходе акцент переносится на конкретные характеристики физической или абстрактной системы, являющейся предметом программного моделирования... Объекты обладают целостностью, которая не должна - а, в действительности, не может - быть нарушена. Объект может только менять состояние, вести себя, управляться или становиться в определенное отношение к другим объектам. Иначе говоря, свойства, которые характеризуют объект и его поведение, остаются неизменными. Например, лифт характеризуется теми неизменными свойствами, что он может двигаться вверх и вниз, оставаясь в пределах шахты... Любая модель должна учитывать эти свойства лифта, так как они входят в его определение».

Появление объектно-ориентированного подхода связано также и со следующими событиями:

* прогресс в области архитектуры ЭВМ;
* развитие языков программирования, таких как **Simula, Smalltalk, Ada**;
* развитие методологии программирования, включая принципы модульности и скрытия данных.

К этому еще следует добавить три момента, которые оказали влияние на становление объектного подхода:

* развитие теории баз данных;
* исследования в области искусственного интеллекта;
* достижения философии и теории познания.

Но наиболее значительный вклад в объектный подход внесен объектными и объектно-ориентированными языками программирования. Впервые понятия классов и объектов введены в языке **Simula 67**. Внесение объектно-ориентированного подхода в **С** привело к возникновению языков **C++.** На основе языка **Pascal** возникли **ObjectPascal** и **Ada**. Появились диалекты **LISP**, с возможностями языков **Simula** и **Smalltalk**.

Первым, кто указал на необходимость построения систем в виде структурированных абстракций, был Дейкстра.

Развивавшиеся достаточно независимо технологии построения баз данных также оказали влияние на объектный подход, в первую очередь благодаря так называемой модели «***сущность-отношение***» (**ER**, entity-relationship). В моделях **ER**, впервые предложенных Ченом, моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений.

Что же такое объектно-ориентированное программирование? Будем понимать его

следующим образом:

***Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.***

В данном определении можно выделить три части:

1. **ООП использует в качестве базовых элементов *объекты,* а не алгоритмы (иерархия «быть частью»).**
2. **Каждый объект является *экземпляром* какого-либо определенного *класса.***
3. **Классы организованы *иерархически*.**

Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к ООП, а называется ***программированием на основе абстрактных типов данных.***

В соответствии с этим определением не все языки программирования являются объектно-ориентированными. Страуструп определил так: «если термин ***объектно-ориентированный язык*** вообще что-либо означает, то он должен означать язык, имеющий средства хорошей поддержки объектно-ориентированного стиля программирования... Обеспечение такого стиля в свою очередь означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле ООП требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям ООП». Теоретически возможна имитация объектно-ориентированного программирования на обычных языках, таких, как **Pascal** и даже **Ассемблер**, но это крайне затруднительно. Другие теоретики проектирования ПО подчеркивают, что: «язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

1. **Поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.**
2. **Объекты относятся к соответствующим типам (классам).**
3. **Типы (классы) могут наследовать атрибуты супертипов (суперклассов)».**

***Абстрагирование*** является одним из основных подходов, используемых для решения сложных задач. Считается, что «абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий». Другие специалисты определяют это понятие как: «Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны». Суммируя разные точки зрения, можно сформулировать следующее определение абстракции:

***Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.***

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных особенностей. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Причем существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование.

Приведем эти абстракции, начиная от наиболее «полезных» к наименее «полезным»:

* ***абстракция сущности*** (объект представляет собой полезную модель некой сущности в предметной области);
* ***абстракция поведения*** (объект состоит из обобщенного множества операций);
* ***абстракция виртуальной машины*** (объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня);
* ***произвольная абстракция*** (объект включает в себя набор операций, не имеющих друг с другом ничего общего).

В первую очередь необходимо стараться строить абстракции сущности, так как они прямо соответствуют сущностям предметной области.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержание. Эти атрибуты являются статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта: файл можно увеличить или уменьшить, изменить его имя и содержимое. В процедурном стиле программирования действия, изменяющие динамические характеристики объектов, составляют суть программы. Любые события связаны с вызовом подпрограмм и с выполнением операторов. Стиль программирования, ориентированный на правила, характеризуется тем, что под влиянием определенных условий активизируются определенные правила, которые в свою очередь вызывают другие правила, и т.д. Объектно-ориентированный стиль программирования связан с воздействием на объекты (***с передачей объектам сообщений***). Так, операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют поведение этого объекта.

**Примеры абстракций.** Для иллюстрации сказанного выше приведем несколько примеров. В данном случае мы сконцентрируем внимание не столько на выделении абстракций для конкретной задачи, сколько на способе выражения абстракций.

В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия или другой почвы. Управление режимом работы парниковой установки – очень ответственное дело, зависящее как от вида выращиваемых культур, так и от стадии выращивания. Нужно контролировать целый ряд факторов: температуру, влажность, освещение, кислотность (показатель рН) и концентрацию питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Попросту говоря, цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения режима выращивания.

Одна из ключевых абстракций в такой задаче – датчик. Известно несколько разновидностей датчиков. Все, что влияет на урожай, должно быть измерено, так что мы должны иметь датчики температуры воды и воздуха, влажности, рН, освещения и концентрации питательных веществ. С внешней точки зрения датчик температуры – это объект, который способен измерять температуру там, где он расположен. Что такое температура? Это числовой параметр, имеющий ограниченный диапазон значений и определенную точность, означающий число. Что такое местоположение датчика? Это некоторое идентифицируемое место в теплице, температуру в котором нам необходимо знать; таких мест, вероятно, немного. Для датчика температуры существенно не столько само местоположение, сколько тот факт, что данный датчик расположен именно в данном месте и это отличает его от других датчиков. Теперь можно задать вопрос о том, каковы обязанности датчика температуры? Мы решаем, что датчик должен знать температуру в своем местонахождении и сообщать ее по запросу. Какие же действия может выполнять по отношению к датчику клиент? Мы принимаем решение о том, что клиент может калибровать датчик и получать от него значение текущей температуры.

Итак, разработчики программного обеспечения нашли механизм борьбы со сложностью – это аб­страгирование, т.е. ***способность отделить логический смысл фрагмента программы от проблемы его реализации***. От процедур к модулям, далее к абстрактным данным и, наконец, к объектам. На этой основе была создана объектно-ориентированная методология и языки программирования, к числу основных преимуществ которых относятся:

* наглядность процесса перехода от бизнес-процессов к программным моделям;
* наличие средств быстрой разработки приложений;
* эффективная организация процесса создания приложений за счет коллективной работы не­скольких специалистов над отдельными частями системы;
* возможность повторного использования кода в различных приложениях и применение компо­нентов, написанных другими разработчиками, что позволяет без больших усилий расширять функциональность программ;
* улучшенные средства взаимодействия с распределенными приложениями и операционными системами;
* функции создания интуитивных графических интерфейсов пользователя.

ООП является лишь последним звеном в цепи решений, которые были предложены для разрешения «кризиса программного обеспечения». Несмотря на то, что ООП действительно помогает в проектировании сложных и больших программ, ООП не панацея. ООП является новым пониманием того, что называется вычислениями. ООП - это не просто набор некоторых свойств, добавленных в уже существующие языки. Это новый шаг в осмыслении задач и разработки **ПО**. ООП – это взгляд на программирование, сосредоточенный на данных, в котором данные и поведе­ние жестко связаны. Программы – это совокупность взаимодействующих объектов. Каждый объ­ект отвечает за конкретную задачу. Вычисление осуществляется посредством взаимодействия объектов.

Чтобы стать специалистом в ООП, недостаточно просто добавить новые знания, необхо­димо полная переоценка привычных методов разработки программ.

Таким образом, любой объект реального мира можно описать с помощью двух наборов:

* атрибутов;
* поведения.

***Атрибут*** – это характеристика объекта. Например, у человека есть имя, фамилия, рост и вес. Имя, фамилия, рост и вес – это атрибуты всех людей. Человека характеризуют и сот­ни других атрибутов, но для конкретной задачи можно остановиться на этих четырех – основных.

***Поведение*** – это действие, которое объект может осуществить. Человек может: сидеть, стоять, идти или бежать, и это не считая тысяч других вариантов поведения.

Нетрудно сообразить, что представляют собой как объекты автомобиль, самолет и даже форма заказа, каждый со своими атрибутами и поведением. Атрибуты и варианты поведения автомобиля и самолета довольно очевидны. Оба они имеют ширину, высоту, вес, колеса и двигатель**,** а также множество других атрибутов. Автомобиль и самолет могутдвигаться в некотором направлении, останавливаться и начинать двигаться в другом направлении, также они могут осуществлять сотни других действий (вариантов поведения). Однако сложнее определить **атрибуты и варианты поведения формы заказа**. **Атрибутами** этого объекта являются**:** имя клиента, адрес клиента, заказанный товар, его количество, а также другая ин­формация, которую можно найти в форме заказа. Варианты поведения формы заказа включают сбор информации, модификацию информации и обработку заказа.

При создании программных систем программисты рассматривают объекты двумя способами:

* как **абстрактные** объекты;
* как **реальные** объекты.

***Абстрактный объект***–это описание реального объекта предметной области минус подробно­сти. Например, абстрактный человек – это описание человека, которое содержит атрибуты и ва­рианты поведения. Вот четыре атрибута, которые могут быть найдены у абстрактного человека (обратите внимание, что эти атрибуты только **идентифицируют** тип характеристики, например имя или вес, но **не определяют само** имя или значение):

* Имя
* Фамилия
* Рост
* Вес

Абстрактный объект используется как модельдля реального объекта конкретной предметной области. Реальный человек имеет все атрибуты и варианты поведения, определенные в абстрактном объекте, а также те подробности, которые были упущенные в абстрактном объекте. Например, абстрактный человек – это модель реального че­ловека. Абстрактный человек определяет, что реальный человек **должен** иметь имя, фамилию, рост и вес. Реальный человек определяет **значения**, связанные с этими атрибутами, например такие:

* Иван
* Петров
* 180 см
* 80 кг

Программисты создают абстрактный объект, а затем используют его для создания реального объекта. Реальный объект называется экземпляром абстрактного объекта. Можно ска­зать, что реальный человек – это **экземпляр** абстрактного человека.

Абстрактные объекты можно трактовать, например, как **формочки** для печенья. Формочка для печенья выглядит как печенье, но печеньем не является.

Почему же целесообразнее смотреть на мир как на множество объектов? – **фокусирование на объектах упрощает для нас понимание сложных вещей**. Объекты позволяют уде­лять внимание важным для нас подробностям и игнорировать те подробности, которые нам неинтересны. Например, преподаватель – это человек, и **он имеет множество атрибутов и вари­антов поведения, которые вам интересны**. Тем не менее, вы, возможно, игнорируете множест­во атрибутов и вариантов поведения преподавателя и фокусируетесь только на тех, которые имеют отношение к вашему образованию.

Аналогично, преподаватель фокусируется на ваших атрибутах и вариантах поведения, которые показывают, как хорошо вы изучаете материал на занятиях. Другие атрибуты, такие как ваша ра­бота, на других занятиях или ваш рост и вес, игнорируются преподавателем.

Таким образом, и вы, и преподаватель упрощаете ваши отношения, решая, какие атрибуты и варианты поведе­ния являются важными для ваших целей, и используя в ваших отношениях только их.

Можно сделать вывод, что объектно-ориентированный подход – это ***способ мышления***, а также способ ***восприятия*** окружающего мира с точки зрения теории объектов. Короче говоря, объектно-ориентированный подход включает в себя все то, что можно описать в терминах теории объектов.

Определение программной системы в терминах объектов – это наиболее понятный способ разработки реального программного обеспечения. Объекты заставляют вас воспринимать все с той точки зрения, **что объ­ект делает, т.е. мысленно моделировать его поведение**. Благодаря этому вы можете отвлечь­ся от рассмотрения объекта с точки зрения того, как он будет реализован в процессе исполнения программы. Таким образом, в процессе написания программы можно использовать естественные термины реального мира. Вместо того чтобы строить программу в форме отдельных процедур и данных, т.е. в терминах мира компьютеров, программа строится из объектов. Причем объекты позволяют в программе моделировать реальный мир с помощью элементов, взятых из предметной области:

* существительных;
* глаго­лов;
* прилагательных.

***Предметная область*** – это **абстрактное пространство**, в котором формулируется определенная задача, т.е. набор понятий, представляющих важные аспекты решаемой задачи

Рассуждая в терминах решаемой задачи, можно избежать опасности увязнуть в деталях реализа­ции. Конечно, некоторые высокоуровневые объекты должны взаимодействовать с компьютером, пользуясь низкоуровневыми, машинно-ориентированными методами. Однако объект изолирует это взаимодействие от остальной части системы.

Объектно-ориентированная программа, как и реальный мир, состоит из объектов. В объектно-ориентированных языках программирования, например, в **VB** и **С#** все является объектами, начиная от самых пер­вичных, базовых типов, целых, логических и до наиболее сложных экземпляров типов пользователя.

Подобно реальным объектам, объекты ООП классифицируются по:

* свойствам
* поведению.

Так, например, в биологии собаки, коты, слоны и люди относятся к млекопитающим. Объединяют этих разных существ общие свойства. Таким же образом в мире программного обеспечения объекты относят­ся к одному или разным классам.

Объектам одного класса присущи общие свойства. Иными словами, класс объектов определяет**:**

* свой­ства
* поведение,
* сообщения (сообщения на которые ***реагирует*** объект) – это когда один объект оказывает влияние на поведение другого объекта, он не делает это непосредственно, а просит другой объект изменить себя, используя некоторую дополнительную информацию.

Объект обнаруживает признаки тогда, когда предоставляет непосредственный доступ к внутрен­ней переменной или возвращает значение с помощью метода.

Поведение – **это действия**, выполняемые объектом **в ответ на сообщение** или на изменение состояния. Это то, что объект делает.

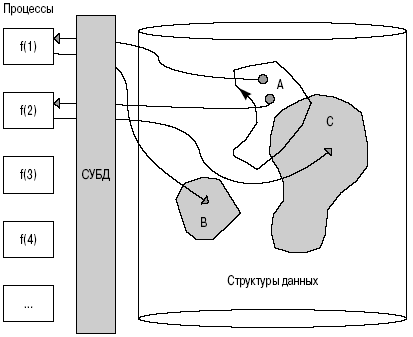
Один объект может влиять на поведение другого объекта, выполняя действия над ним. Вместо термина действие употребляют термины: ***вызов метода, вызов функции или передача сообщения***. Важно, конечно, не то, какой из этих терминов употребляется, а то, что эти действия вызывают проявление поведения объекта.

Термин ***передача сообщения*** наиболее близко передает суть объектно-ориентированного под­хода. Этот термин отражает динамичность взаимодействия объектов. Он подчеркивает разницу между сообщением и объектом. С его помощью легче понять суть взаимодействия объектов.

На практике, в основном применяется термин ***вызов метода***. Он может быть взаимозаменяем с термином ***сообщение***.

С другой стороны **ООП** – это иной способ осмысления, формулирования и решения задач по созданию программ. У программирования на объектно-ориентированном языке масса преимуществ, причем это относится не только к созданию более эффективного кода, но и к модификации и расширению возможностей уже имеющихся систем. Многим поначалу такое утверждение не кажется столь очевидным.

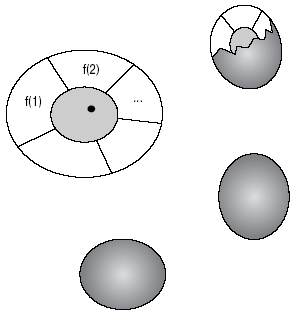
Традиционная компьютерная система, основанная на так называемой аппаратной архитектуре фон Неймана, может рассматриваться как множество функций (процедур) или процессов, работающих с набором данных. Эта статическая архитектурная модель показана на рис. 3.1-5.

  
Рис. 3.1-5. Архитектура обычной компьютерной системы

Из рисунка видно, что в процессе работы системы динамика состоит в вызове некоторой процедуры **f(1)**, которая считывает соответствующие данные **A**, преобразовывает их и записывает в **B**. Потом вызывается некоторая другая процедура **f(2)**, которая тоже считывает некоторые данные (возможно, те же), использует их по назначению и записывает в **C**. Подобный перекрывающийся доступ к данным порождает проблемы параллельной обработки и целостности, которые могут быть разрешены с помощью систем управления доступом к данных. Перед тем как двигаться дальше, ст*о*ит рассмотреть вопрос, что нужно сделать, чтобы изменить часть структуры данных.

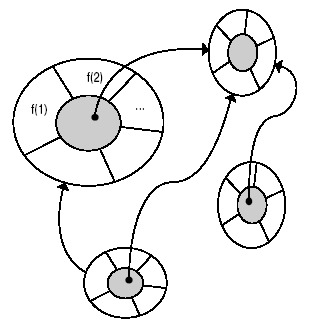
Рассматривая это с точки зрения специалиста по поддержке программы, можно сделать единственный вывод: необходимо проверить, затрагивают ли эти изменения каждую отдельную процедуру. В этом вопросе может помочь качественная документация, но на практике она доступна редко. Отчасти это объясняется тем, что хорошая документация сама должна состоять из объектно-ориентированного описания системы, и ее невозможно представить в отрыве от объектно-ориентированной реализации или, по крайней мере, проектного решения. Кроме того, изменение каждой процедуры в соответствии с новыми структурами данных может иметь побочный эффект и в других частях системы. Возможно, это объясняет необычайно высокую стоимость поддержки программных систем.

На рис. 3.1-6 показан совершенно иной архитектурный подход к системам.

  
Рис. 3.1-6. Архитектура объектно-ориентированной системы

Все данные, доступ к которым нужен процедурам, инкапсулируются в один пакет с этой функцией – ***объект****,*  таким образом, чтобы другой объект не имел доступа к этим данным. Можно рассматривать эти объекты как яйца. Желток – это структура данных, белки это функции, которые имеют доступ к этим данным, а скорлупа представляет сигнатуру общедоступных операций. Оболочка интерфейса скрывает реализацию и самих функций, и структур данных. Предположим, что в яйце, изображенном «в разрезе» на   
рис. 3.1-6, изменилась структура данных. Тогда специалисты должны проверить только факт влияния изменений на белок этого яйца; сфера вмешательства локализована. Изменение реализации одного объекта не может затронуть “внутренность” другого. В этом и состоит **инкапсуляция**: данные и процессы их обработки **объединяются** и **скрываются** за интерфейсом.

Однако при использовании этой модели в чистом виде возникает проблема. Допустим, каждый объект содержит функции, которым нужны одинаковые данные. В таком случае появляется необходимость дублирования данных, и подход становится довольно непрактичным. Решение состоит в передаче сообщений между объектами. Тогда объект **X** сможет использовать данные **A**, но не инкапсулировать их. При условии, что в желтке **X** хранится идентичность другого объекта **Y**, который содержит необходимые данные, он может передать сообщение, запрашивающее эти данные или даже несколько преобразованную их версию. Это показано на рис. 3.1-7, где маленькая черная точка символизирует идентичность целевого объекта, а стрелки показывают направление передачи сообщения. Можно сказать, что это – половина принципа объектной технологии. Вторая половина – это возможность классификации объектов и их связывания различными способами. Обратите внимание на то, что этот подход локализует и таким образом сильно упрощает исходную задачу.

****Рис. 3.1-7. Передача сообщений исключает дублирование данных

При изменении структуры данных программисту нужно всего лишь проверить функции в белке яйца, инкапсулирующего эти данные. Это изменение не может повлиять на другие части системы, если оболочка не повреждается или не деформируется, т.е. если не меняется интерфейс. Таким образом, если необходимо на порядок уменьшить проблемы сопровождения, нужно очень упорно поработать, чтобы обеспечить корректные, полные и устойчивые интерфейсы объектов. Из этого следует, что в данном случае согласованный анализ и проектирование должны быть более эффективны и необходимы, чем для традиционных систем. Этот дополнительный объем работ дает результат, потому что объектная технология ведет к некоторым очень существенным преимуществам.

### 3.3.2. Объектно-ориентированная декомпозиция и объектная модель

Проблемы, стимулировавшие развитие объектно-ориентированного подхода (ООП):

* необходимость повышения производительности разработки за счет многократного (повторного) использования ПО;
* необходимость упрощения сопровождения и модификации разработанных систем (локализация вносимых изменений);
* облегчение проектирования систем (за счет сокращения семантического разрыва между структурой решаемых задач и структурой ПО).

Забегая вперед, скажем, какие решения данных проблем дает ООП. При ООП изменения локализуются внутри класса (компоненты или пакета, если изменяются несколько классов). Семантический разрыв ликвидируется, поскольку сущности предметной области представляются объектами, следовательно, разработчик и заказчик (пользователь) оперируют схожими понятиями. Повторное использование достигается за счет построения систем с использованием библиотек готовых компонент – модулей (заимствовано из структурного или функционального подхода).

В основе объектно-ориентированного подхода лежит объектная декомпозиция, при этом статическая структура ПО описывается в терминах объектов и связей между ними, а динамический аспект ПО описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

Объектная модель является естественным способом представления реального мира. Она является концептуальной основой ООП. ***Основными принципами*** ее построения являются:

* абстрагирование;
* инкапсуляция;
* модульность;
* иерархия.

Дополнительные принципы:

* типизация;
* параллелизм;
* устойчивость (persistence).

***Абстрагирование*** – это выделение наиболее существенных характеристик некоторого объекта, отличающих его от всех других видов объектов, важных с точки зрения дальнейшего рассмотрения и анализа, и игнорирование менее важных или незначительных деталей. Абстракцией является любая модель, включающая наиболее важные, существенные или отличительные характеристики некоторого объекта, и игнорирующая менее важные или незначительные детали. Абстрагирование позволяет управлять сложностью системы, концентрируясь на существенных свойствах объекта. Абстракция зависит от предметной области и точки зрения – то, что важно в одном контексте, может быть неважно в другом. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Объекты и классы – основные абстракции предметной области.

***Инкапсуляция*** – локализация свойств и поведения в рамках единственной абстракции (рассматриваемой как «черный ящик»), скрывающей реализацию за общедоступным интерфейсом. При инкапсуляции отделяется внутреннее устройство объекта от его внешнего поведения. Объектный подход предполагает, что внутренние ресурсы объекта, скрыты от внешней среды. Абстрагирование и инкапсуляция являются взаимодополняющими принципами.

***Модульность*** – это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (частей). Модульность снижает сложность системы, позволяя выполнять независимую разработку ее отдельных частей.

***Иерархия*** – ранжированная или упорядоченная система абстракций, расположение их по уровням в виде древовидной структуры. Элементы, находящиеся на одном уровне иерархии, должны также находиться на одном уровне абстракции. Основными видами иерархических структур сложных систем являются структура классов и структура объектов. Иерархия классов строится по наследованию, а иерархия объектов – по агрегации.

***Тип*** – точная характеристика некоторой совокупности однородных объектов, включающая структуру и поведение.

***Типизация*** – способ защититься от использования объектов одного класса вместо другого, или, по крайней мере, управлять таким использованием.

При строгой типизации запрещается использование объектов неверного типа, требуется явное преобразование к нужному типу. При менее строгой типизации такого рода запреты ослаблены. В частности, допускается полиморфизм – многозначность имен. Одно из проявлений полиморфизма, использование объект подтипа (наследника) в роли объекта супертипа (предка).

***Параллелизм*** – наличие в системе нескольких потоков управления одновременно. Объект может быть активен, т. е. может порождать отдельный поток управления.Различные объекты могут быть активны одновременно.

***Устойчивость*** – способность объекта сохранять свое существование во времени и/или пространстве (адресном, в частности при перемещении между узлами вычислительной системы).

Рассмотрим основные понятия объектно-ориентированного подхода (***элементы объектной модели*).** К ним относятся: **объект; класс; атрибут; операция; полиморфизм; наследование; компонент; пакет; подсистема; связь**.

***Объект*** – осязаемая сущность (tangibleentity) – предмет или явление (процесс), имеющие четко выраженные границы, индивидуальность и поведение. Любой объект обладает состоянием, поведением и индивидуальностью. ***Состояние объекта*** определяется значениями его свойств (атрибутов) и связями с другими объектами, оно может меняться со временем. ***Поведение*** определяет действия объекта и его реакцию на запросы от других объектов. Поведение представляется с помощью набора сообщений, воспринимаемых объектом (операций, которые может выполнять объект).

***Индивидуальность*** – это свойства объекта, отличающие его от всех других объектов.

Структура и поведение схожих объектов определяют общий для них класс. ***Класс*** – это множество объектов, связанных общностью свойств, поведения, связей и семантики. Любой объект является экземпляром класса. Определение классов и объектов – одна из самых сложных задач объектно-ориентированного проектирования.

***Атрибут*** – поименованное свойство класса, определяющее диапазон допустимых значений, которые могут принимать экземпляры данного свойства. Атрибуты могут быть скрыты от других классов, это определяет видимость атрибута: рublic (общий, открытый); private (закрытый, секретный); protected (защищенный).

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется операцией или посылкой сообщения. ***Операция*** – это реализация услуги, которую можно запросить у любого объекта данного класса. Операции реализуют связанное с классом поведение, его обязанности. Описание операции включает четыре части: имя; список параметров; тип возвращаемого значения; видимость.

Результат операции зависит от текущего состояния объекта. Виды операций:

* операции реализации (implementoroperations) – реализуют требуемую функциональность;
* операции управления (manageroperations) управляют созданием и уничтожением объектов (конструкторы и деструкторы);
* операции доступа (access operations) – так называемые, get-теры, set-теры – дают доступ к закрытым атрибутам;
* вспомогательные операции (helperoperations) – непубличные операции, служат для реализации операций других видов.

Понятие полиморфизма может быть интерпретировано, как способность объекта принадлежать более чем одному типу. ***Полиморфизм*** – это способность скрывать множество различных реализаций под единственным общим интерфейсом.

***Интерфейс*** – это совокупность операций, определяющих набор услуг класса или компонента. Интерфейс не определяет внутреннюю структуру, все его операции открыты. Например, одна и та же операция *рассчитать Зарплату* может иметь три различные реализации в трех различных классах: СлужащийСПочасовойОплатой, СлужащийНаОкладе, ВременныйСлужащий.

***Наследование*** – это построение новых классов на основе существующих с возможностью добавления или переопределения свойств (атрибутов) и поведения (операций).

***Компонент*** – это относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры.

Компонент представляет собой физическую реализацию проектной абстракции и может быть: компонентом исходного кода (.cpp); компонентом времени выполнения (.dll, ActiveX и т. п.); исполняемый компонентом (.exe). Компонент обеспечивает физическую реализацию набора интерфейсов. Компонентная разработка (component-based-development) представляет собой создание программных систем, состоящих из компонентов (не путать с объектно-ориентированным программированием ООП – способ создания программных компонентов, базирующихся на объектах).

Компонентная разработка – технология, позволяющая объединять объектные компоненты в систему.

***Пакет*** – это общий механизм для организации элементов в группы. Это элемент модели, который может включать другие элементы. Каждый элемент модели может входить только в один пакет. Пакет является:

* средством организации модели в процессе разработки, повышения ее управляемости и читаемости;
* единицей управления конфигурацией.

***Подсистема*** – это комбинация пакета (может включать другие элементы модели) и класса (обладает поведением). Подсистема реализует один или более интерфейсов, определяющих ее поведение. Она используется для представления компонента в процессе проектирования.

Между элементами объектной модели существуют различные виды ***связей***:

* **ассоциация** – это семантическая (структурная) связь между классами (пример: компания-наниматель имеет ассоциативную связь с персоной-работником);
* **агрегация** – более сильный тип ассоциативной связи между целым и его частями (пример: автомобиль и мотор);
* **композиция** – усиленная агрегация, когда часть не может существовать без целого (пример: университет, факультет, кафедра);
* **зависимость** – связь между двумя элементами модели, при которой изменения в спецификации одного элемента могут повлечь за собой изменения в другом элементе;
* **обобщение** – связь «тип – подтип»;
* **реализаци**я – семантическая связь между двумя классификаторами (один из классификаторов является контрактом, который другой обязуется выполнять), такова связь между интерфейсом и реализующим его классом, интерфейсом и реализующей его подсистемой, вариантом использования и реализующей его кооперацией.

Связи характеризуются: направлением, именем и ролевыми именами участников связи, мощностью.

***Мощность*** (multiplicity) показывает, как много объектов участвует в связи. Мощность – это количество объектов одного класса, связанных с ОДНИМ объектом другого класса. Для каждой связи ассоциации и агрегации может существовать два указателя мощности – по одному на каждом конце связи.

***Наследование*** – построение новых классов на основе существующих с возможностью добавления или переопределения свойств (атрибутов) и поведения (операций). Наследование – это отношение между суперклассом и его подклассами.

Два пути определения наследования: обобщение и специализация.

Общие атрибуты, операции и/или отношения отображаются на верхнем уровне иерархии. В объектной модели наследование может быть множественным.

На связи могут накладываться ограничения. Например, если необходимо, множественное наследование в некоторой иерархии классов может быть запрещено.

Подведем итоги.

Способ управления сложными системами был известен еще в древности – «**разделяй и властвуй**». При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае для понимания любого уровня системы нам необходимо одновременно держать в уме информацию лишь о немногих ее частях. В самом деле, декомпозиция вызвана сложностью программирования системы, поскольку именно эта сложность вынуждает делить пространство состояний этой программной системы.

***Алгоритмическая декомпозиция*.** Большинство специалистов, которые при создании программных систем используют технологию структурного программирования, и проектированию «сверху вниз», в частности, программисты воспринимают декомпозицию как обычное разделение алгоритмов на модули или процедуры, где каждый модуль системы выполняет один из этапов общего процесса.

***Объектно-ориентированная декомпозиция*.** Здесь происходит разделение модели предметной области на элементы, выбрав в качестве критерия декомпозиции принадлежность этих элементов к различным абстракциям данной рассматриваемой предметной области. Прежде чем разделять задачу на шаги, необходимо определить какие объекты предметной области будут входить в разрабатываемую модель.

Хотя обе схемы решают одну и ту же задачу, но они делают это разными способами и при этом используют различные модели. Во второй случае мир представляется совокупностью автономных элементов предметной области, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. Каждый элемент обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует вполне определенное поведение конкретной части предметной области. Объекты что-то делают, и им можно послать сообщение, попросив их выполнить то-то и то-то. Так как данная декомпозиция основана на объектах, а не на алгоритмах, она называется ***объектно-ориентированной декомпозицией.***

***Декомпозиция: алгоритмическая или объектно-ориентированная*?** А какая декомпозиция сложной программной системы правильнее – по алгоритмам или по объектам? Правильный ответ на вопрос: важны оба аспекта. Разделение по алгоритмам концентрирует внимание на порядке ***происходящих событий и вычислительном процессе***, а разделение по объектам придает особое значение элементам, которые являются ***объектами***. Однако нельзя сконструировать сложную систему одновременно двумя способами. Разделение системы возможно либо по объектам, либо по алгоритмам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения.

Опыт специалистов показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться со сложностью программных систем. Кроме того, объектная декомпозиция имеет несколько важных преимуществ перед алгоритмической декомпозицией. Объектная декомпозиция уменьшает размер программных систем за счет повторного использования общих механизмов, что приводит к существенной экономии выразительных средств. Объектно-ориентированные системы более гибки и проще эволюционируют со временем, потому что их схемы базируется на устойчивых промежуточных формах. Действительно, объектная декомпозиция существенно снижает риск при создании сложной программной системы, так как она развивается из меньших систем, в которых мы уже уверены. Более того, объектная декомпозиция помогает на ранних стадиях проектирования разобраться в сложной программной системе.

***Важность построения моделей*.** Моделирование широко распространено во всех инженерных дисциплинах, в значительной степени из-за того, что оно реализует принципы ***декомпозиции, абстракции*** и ***иерархии***. Каждая модель описывает определенную часть рассматриваемой системы, далее строятся новые модели на базе старых моделей и д. т. Модели позволяют нам контролировать наши неудачи. Мы оцениваем поведение каждой модели в обычных и необычных ситуациях, а затем проводим соответствующие доработки, если нас что-то не удовлетворяет.

Чтобы понять во всех тонкостях поведение сложной системы, приходится использовать не одну модель. Например, проектируя компьютер на одной плате, инженер-электронщик должен рассматривать систему как на уровне отдельных элементов схемы (микросхем), так и на уровне схемы. Схема помогает инженеру разобраться в совместном поведении микросхем. Схема представляет собой план физической реализации системы микросхем, в которой учтены размер платы, потребляемая мощность и типы имеющихся интегральных микросхем. С этой точки зрения инженер может независимо оценивать такие параметры системы, как температурное распределение и технологичность изготовления. Проектировщик платы может также рассматривать динамические и статические особенности системы.

***Элементы методологии проектирования ПО*.** Ясно, что не существует такого универсального метода, который бы провел программиста по пути от требований к сложной программной системе до их выполнения. Проектирование сложной программной системы отнюдь не сводится к слепому следованию некоему набору рецептов. Скорее это постепенный и итеративный процесс. И, тем не менее, использование методологии проектирования вносит в процесс разработки определенную организованность. Программисты разработали десятки различных методов, которые мы можем классифицировать по трем категориям. Несмотря на различия, эти методы имеют что-то общее. Их, в частности, объединяет следующее:

* условные обозначения – язык для описания каждой модели;
* процесс – правила проектирования модели;
* инструменты – средства, которые ускоряют процесс создания моделей, и в которых уже воплощены законы функционирования моделей.

Хороший метод проектирования базируется на прочной теоретической основе и при этом дает программисту известную степень свободы самовыражения.

***Объектно-ориентированные модели.*** Существует ли наилучший способ декомпозиции сложной системы? Если и существует, то пока он никому не известен. Этот вопрос можно поставить следующим образом: Как наилучшим способом разделить сложную систему на подсистемы? Еще раз необходимо напомнить, что полезнее всего создавать такие модели, которые фокусируют внимание на объектах, заданной предметной области, и которые образуются в результате ***объектно-ориентированной декомпозиции.***

***Объектно-ориентированный анализ и проектирование*** – это метод, логически приводящий нас к объектно-ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем гибкие программы, написанные экономными средствами. При разумном разделении пространства состояний мы добиваемся большей уверенности в правильности нашей программы. В итоге, мы уменьшаем риск при разработке сложных программных систем.

***Объектно-ориентированная технология*** основывается на так называемой ***объектной модели****.* Основными ее принципами являются:

* ***абстрагирование;***
* ***инкапсуляция;***
* ***модульность;***
* ***иерархичность;***
* ***типизация;***
* ***параллелизм;***
* ***сохраняемость (персистентность)***.

Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования. Эти отличия обусловлены тем, что структурное проектирование основано на структурном программировании, тогда как в основе объектно-ориентированного проектирования лежит методология объектно-ориентированного программирования.

Выясним, чем же является и чем не является объектно-ориентированная разработка программ, и в чем отличия этого подхода к проектированию от других с учетом семи перечисленных выше принципов объектной модели.

В большинстве своем программисты используют в работе один язык программирования и следуют одному стилю. Они программируют в парадигме, навязанной используемым ими языком. Часто они оставляют в стороне альтернативные подходы к цели, а, следовательно, им трудно увидеть преимущества стиля, более соответствующего решаемой задаче. Ранее, в Теме 3.2 стиль программирования был определен как ***способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле***. Кроме того были перечислены пять основных стилей программирования, которые вместе с присущими им видами абстракций:

* процедурно-ориентированный стиль (алгоритмы);
* объектно-ориентированный стиль (классы и объекты);
* стиль, ориентированный на логико-ориентированные цели (исчисления предикатов);
* стиль, ориентированный на правила (если-то);
* стиль, ориентированный на ограничения (инвариантные соотношения).

Невозможно признать какой-либо стиль программирования наилучшим во всех областях практического применения. Например, для проектирования баз знаний более пригоден стиль, ориентированный на правила, а для вычислительных задач - процедурно-ориентированный. По мнению специалистов, объектно-ориентированный стиль является наиболее приемлемым для широчайшего круга приложений; действительно, эта парадигма часто служит архитектурным фундаментом для других парадигм.

Каждый стиль программирования имеет свою концептуальную базу и способ восприятия решаемой задачи.

### 3.3.3. Основные принципы моделирования программного обеспечения

Центральным элементом деятельности, ведущей к созданию первоклассного программного обеспечения, является моделирование. Модели позволяют нам наглядно продемонстрировать желаемую структуру и поведение системы. Они также необходимы для ***визуализации и управления ее архитектурой***. Модели помогают добиться лучшего понимания создаваемой нами системы, что зачастую приводит к ее упрощению и возможности повторного использования. Наконец, модели нужны для минимизации риска.

Если вы хотите соорудить собачью конуру, то можете приступить к работе, имея в наличии лишь кучу досок, горсть гвоздей, молоток, плоскогубцы и рулетку. Несколько часов работы после небольшого предварительного планирования – и вы, надо полагать, сколотите вполне приемлемую конуру, причем, скорее всего, без посторонней помощи. Если конура получится достаточно большой и не будет сильно протекать, собака останется довольна.

Если вам надо построить дом для своей семьи, вы, конечно, можете воспользоваться тем же набором инструментов, но времени на это уйдет значительно больше, и ваши домочадцы, надо полагать, окажутся более требовательными, чем собака. Если у вас нет особого опыта в области строительства, лучше тщательно все продумать перед тем, как забить первый гвоздь. Стоит, по меньшей мере, сделать хотя бы несколько эскизов внешнего вида будущей постройки. Без сомнения, вам нужно качественное жилье, удовлетворяющее запросам вашей семьи и не нарушающее местных строительных норм и правил, – а значит, придется сделать кое-какие чертежи с учетом назначения каждой комнаты и таких деталей, как освещение, отопление и водопровод. На основании этих планов вы сможете правильно рассчитать необходимое для работы время и выбрать подходящие стройматериалы. В принципе можно построить дом и самому, но гораздо выгоднее прибегнуть к помощи других людей, нанимая их для выполнения ключевых работ или покупая готовые детали. Коль скоро вы следуете плану и укладываетесь в смету, ваша семья будет довольна. Если же что-то не сладится, вряд ли стоит менять семью – лучше своевременно учесть пожелания родственников.

Задавшись целью построить небоскреб для офиса, было бы совсем неразумно браться за дело, имея в распоряжении груду досок и молоток. Поскольку в этом случае, скорее всего, будут привлекаться многочисленные капиталовложения, инвесторы потребуют, чтобы вы учли их пожелания касательно размера, стиля и формы строения. Кстати, они могут изменить свое решение уже после того, как вы приступили к строительству. Так как любой просчет слишком дорого вам обойдется, значение планирования многократно возрастает. По-видимому, вы окажетесь членом большого коллектива проектировщиков и застройщиков, и для успешного взаимодействия вам потребуется делать множество разнообразных чертежей и моделей здания.

Подобрав нужных людей, подходящие инструменты и приступив к реализации намеченного плана, вы с большой вероятностью сумеете построить здание, удовлетворяющее всем требованиям будущих обитателей. Если вы и дальше собираетесь возводить дома, то придется искать компромисс между пожеланиями заказчика и возможностями современной технологии строительства, а к рабочему коллективу относиться с заботой, не рискуя людьми и не требуя работы, отнимающей все силы без остатка.

Хотя это и кажется комичным, многие компании, разрабатывающие программное обеспечение, хотят создать небоскреб, в то время как их подход к делу очень напоминает строительство собачьей конуры.

Конечно, иногда вам может просто повезти. Если нужные люди оказались в нужном месте в нужный час и расположение планет благоприятствует новым начинаниям, то, возможно, вам посчастливится создать продукт, который придется потребителям по душе. Но, скорее всего, нанять подходящих работников не удастся (они все уже заняты), благоприятный момент вы упустите (вчера было бы лучше), а расположение планет будет крайне неудачным (этот фактор вообще неподвластен вашему контролю). В эпоху расцвета Internet, когда спрос на программистов постоянно растет, коллективы разработчиков часто ограничиваются тем единственным, что они по-настоящему умеют делать, - написанием все новых строк кода. Героические усилия, затрачиваемые на программирование, стали легендой в этой отрасли, и кажется, что единственным ответом на трудности в разработке ПО послужит еще более интенсивная работа. Однако написанный код вовсе не обязательно поможет решить проблему, а проект может быть столь грандиозным, что даже увеличения рабочего дня на несколько часов окажется недостаточно для его успешного завершения.

Если вы действительно желаете создать программный продукт, сопоставимый с жилым домом или небоскребом, то задача не сводится к написанию большого объема кода. На самом деле проблема в том, чтобы написать правильный код, причем так, чтобы его объем был минимален. При таком подходе разработка качественного программного обеспечения сводится к вопросам выбора архитектуры, подходящих инструментов и средств управления процессом. Кстати, нужно иметь в виду, что многие проекты, задуманные «по принципу конуры», быстро вырастали до размеров небоскреба, становясь жертвой собственного успеха. Если такой рост не был учтен в архитектуре приложения, технологическом процессе или при выборе инструментов, то неизбежно наступит время, когда выросшая до размеров огромного дома конура обрушится под тяжестью собственного веса. Но если разлетится в щепки конура, это лишь разозлит вашу собаку, а если рухнет небоскреб, это затронет материальные интересы его арендаторов.

Неудачные проекты заканчиваются крахом в силу самых разных причин, а вот успешные, как правило, имеют много общего. Хотя успех программного проекта обеспечивается множеством разных слагаемых, одним из общих является применение моделирования.

Моделирование – это устоявшаяся и повсеместно принятая инженерная методика. Мы строим архитектурные модели зданий, чтобы помочь их будущим обитателям во всех подробностях представить себе готовый продукт. Иногда прибегают даже к математическому моделированию зданий, чтобы учесть влияние сильного ветра или землетрясения.

Моделирование применяется не только в строительстве. Вряд ли вы сумеете наладить выпуск новых самолетов или автомобилей, не испытав свой проект на моделях: от компьютерных моделей и чертежей до физических моделей в аэродинамической трубе, а затем и полномасштабных прототипов. Электрические приборы, от микропроцессоров до телефонных коммутаторов, также требуют моделирования для лучшего понимания системы и организации общения с коллегами. Даже в кинематографии успех фильма невозможен без предварительно написанного сценария (тоже своеобразная форма модели). В социологии, экономике или менеджменте мы также прибегаем к моделированию, которое позволяет проверить наши теории и испытать новые идеи с минимальным риском и затратами.

**Итак, что же такое модель**? Попросту говоря, она является **упрощенным представлением реальности.** Модель – это визуальное отображение системы (чертеж системы): в нее может входить как детальный план, так и более абстрактное представление системы «с высоты птичьего полета». Хорошая модель всегда включает элементы, существенно влияющие на результат, и не включает те, которые малозначимы на данном уровне абстракции. Каждая система может быть описана с разных точек зрения, для чего используются различные модели, каждая из которых, следовательно, является семантически замкнутой абстракцией системы. Модель может быть структурной, подчеркивающей организацию системы, или поведенческой, то есть отражающей ее динамику.

**Зачем мы моделируем**? На это есть фундаментальная причина. ***Мы строим модели для того, чтобы лучше понимать разрабатываемую систему*.**

Моделирование позволяет решить четыре различных задачи:

* визуализировать систему в ее текущем или желательном для нас состоянии;
* определить структуру или поведение системы;
* получить шаблон, позволяющий затем сконструировать систему;
* документировать принимаемые решения, используя полученные модели.

Моделирование предназначено не только для создания больших систем. Даже программный эквивалент собачьей конуры выиграет от этого процесса. Чем больше и сложнее система, тем большее значение приобретает моделирование при ее разработке. Дело в том, что ***моделировать сложную систему необходимо, поскольку иначе мы не можем воспринять ее как единое целое****.*

Восприятие человеком сложных сущностей ограничено. Моделируя, мы сужаем проблему, заостряя внимание в данный момент только на одном аспекте. В сущности, этот подход есть не что иное, как принцип «разделяй и властвуй», который Э. Дейкстра провозгласил много лет назад: сложную задачу легче решить, если разделить ее на несколько меньших. Кроме того, моделирование усиливает возможности человеческого интеллекта, поскольку правильно выбранная модель дает возможность создавать проекты на более высоких уровнях абстракции.

Сказать, что моделирование имеет смысл, еще не означает, что оно абсолютно необходимо. И действительно, многие исследования показывают, что в большинстве компаний, разрабатывающих программное обеспечение, моделирование применяют редко или вовсе не применяют. Чем проще проект, тем менее вероятно, что в нем будет использовано формальное моделирование.

Ключевое слово здесь – «формальное». На практике даже при реализации простейшего проекта разработчики в той или иной мере применяют моделирование, хотя бы неформально. Для визуализации части системы ее проектировщик может нарисовать что-то на доске или на клочке бумаги. Коллектив авторов вправе использовать карточки, чтобы проработать сценарий или конструкцию механизма. Ничего плохого в таких моделях нет. Если они работают, их существование вполне оправдано. Но эти неформальные модели часто создаются для однократного применения и не обеспечивают общего языка, который был бы понятен другим участникам проекта. В строительстве существует общий, понятный для всех язык чертежей; имеются общие языки в электротехнике и математическом моделировании. Организация разработчиков также может извлечь выгоду от использования общего языка для моделирования программного обеспечения.

От моделирования может выиграть любой проект. Даже при создании одноразовых программ, когда зачастую бывает полезнее выбросить неподходящий код из-за преимущества в скорости разработки, которые дают языки визуального программирования, моделирование поможет коллективу разработчиков лучше представить план системы, а значит, выполнить проект быстрее и создать именно то, что подразумевал изначальный замысел. Чем сложнее проект, тем более вероятно, что из-за отсутствия моделирования он потерпит неудачу или будет создано не то, что нужно. Все полезные и интересные системы с течением времени обычно усложняются. Пренебрегая моделированием в самом начале создания системы, вы, возможно, горько пожалеете об этом, когда будет уже слишком поздно.

Моделирование имеет богатую историю во всех инженерных дисциплинах. Длительный опыт его использования позволил сформулировать четыре основных принципа.

**Во-первых**, **выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение**. Иначе говоря, подходите к выбору модели вдумчиво. Правильно выбранная модель высветит самые коварные проблемы разработки и позволит проникнуть в самую суть задачи, что при ином подходе было бы попросту невозможно. Неправильная модель заведет вас в тупик, поскольку внимание будет заостряться на несущественных вопросах.

Можно с уверенностью сказать, что взгляд на мир существенно зависит от выбираемой модели. Если смотреть на систему глазами разработчика баз данных, то основное внимание будете уделять моделям "сущность-связь", где поведение инкапсулировано в хранимых процедурах. Структурный аналитик, скорее всего, создал бы модель, в центре которой находятся алгоритмы и передача данных от одного процесса к другому. Результатом труда разработчика, пользующегося объектно-ориентированным методом, будет система, архитектура которой основана на множестве классов и образцах взаимодействия, определяющих, как эти классы действуют совместно. Любой из этих вариантов может оказаться подходящим для данного приложения и методики разработки, хотя опыт подсказывает, что объектно-ориентированная точка зрения более эффективна при создании гибких архитектур, даже если система должна будет работать с большими базами данных или производить сложные математические расчеты. При этом надо учитывать, что различные точки зрения на мир приводят к созданию различных систем, со своими преимуществами и недостатками.

**Второй принцип формулируется так: *каждая модель может быть воплощена с разной степенью абстракции*.**

При строительстве небоскреба может возникнуть необходимость показать его с высоты птичьего полета, например, чтобы с проектом могли ознакомиться инвесторы. В других случаях, наоборот, требуется самое детальное описание – допустим, чтобы показать какой-нибудь сложный изгиб трубы или необычный элемент конструкции.

То же происходит и при моделировании программного обеспечения. Иногда простая и быстро созданная модель пользовательского интерфейса – самый подходящий вариант. В других случаях приходится работать на уровне битов, например, когда вы специфицируете межсистемные интерфейсы или боретесь с узкими местами в сети. В любом случае лучшей моделью будет та, которая позволяет выбрать уровень детализации в зависимости от того, кто и с какой целью на нее смотрит. Для аналитика или конечного пользователя наибольший интерес представляет вопрос «что», а для разработчика – вопрос «как». В обоих случаях необходима возможность рассматривать систему на разных уровнях детализации в разное время.

**Третий принцип: *лучшие модели - те, что ближе к реальности.***

Физическая модель здания, которая ведет себя не так, как изготовленная из реальных материалов, имеет лишь ограниченную ценность. Математическая модель самолета, для которой предполагаются идеальные условия работы и безупречная сборка, может и не обладать некоторыми характеристиками, присущими настоящему изделию, что в ряде случаев приводит к фатальным последствиям. Лучше всего, если ваши модели будут во всем соотноситься с реальностью, а там, где связь ослабевает, должно быть понятно, в чем заключается различие и что из этого следует. Поскольку модель всегда упрощает реальность, задача в том, чтобы это упрощение не повлекло за собой какие-либо существенные потери.

Возвращаясь к программному обеспечению, можно сказать, что "ахиллесова пята" структурного анализа – несоответствие принятой в нем модели и модели системного проекта. Если этот разрыв не будет устранен, то поведение созданной системы с течением времени начнет все больше отличаться от задуманного. При объектно-ориентированном подходе можно объединить все почти независимые представления системы в единое семантическое целое.

**Четвертый принцип заключается в том, что *нельзя ограничиваться созданием только одной модели. Наилучший подход при разработке любой нетривиальной системы*** – ***использовать совокупность нескольких моделей, почти независимых друг от друга*.**

Если вы конструируете здание, то никакой отдельный комплект чертежей не поможет вам прояснить до конца все детали. Понадобятся, как минимум, поэтажные планы, виды в разрезе, схемы электропроводки, центрального отопления и водопровода.

Ключевым определением здесь является «почти независимые». В данном контексте оно означает, что модели могут создаваться и изучаться по отдельности, но вместе с тем остаются взаимосвязанными. Например, можно изучать только схемы электропроводки проектируемого здания, но при этом наложить их на поэтажный план пола и даже рассмотреть совместно с прокладкой труб на схеме водоснабжения.

Такой подход верен и в отношении объектно-ориентированных программных систем. Для понимания архитектуры подобных систем требуется несколько взаимодополняющих «видов»:

* вид с точки зрения прецедентов, или вариантов использования (чтобы выявить требования к системе); вид с точки зрения проектирования (чтобы построить словарь предметной области и области решения);
* вид с точки зрения процессов (чтобы смоделировать распределение процессов и потоков в системе), вид с точки зрения реализации, позволяющий рассмотреть физическую реализацию системы;
* вид с точки зрения развертывания, помогающий сосредоточиться на вопросах системного проектирования.

Каждый из перечисленных видов имеет множество структурных и поведенческих аспектов, которые в своей совокупности составляют детальный чертеж программной системы.

В зависимости от природы системы некоторые модели могут быть важнее других. Так, при создании систем для обработки больших объемов данных более важны модели, обращающиеся к точке зрения статического проектирования. В приложениях, ориентированных на интерактивную работу пользователя, на первый план выходят представления с точки зрения статических и динамических прецедентов. В системах реального времени наиболее существенными будут представления с точки зрения динамических процессов. Наконец, в распределенных системах, таких как Web-приложения, основное внимание нужно уделять моделям реализации и развертывания.

Инженеры-строители создают огромное количество моделей. Чаще всего это ***структурные*** модели, позволяющие визуализировать и специфицировать части системы и то, как они соотносятся друг с другом. Иногда, в особо критичных случаях, создаются также ***динамические*** модели – например, если надо изучить поведение конструкции при землетрясении. Эти два типа различаются по организации и по тому, на что в первую очередь обращается внимание при проектировании.

При разработке программного обеспечения тоже существует несколько подходов к моделированию. Важнейшие из них – алгоритмический и объектно-ориентированный подход.

***Алгоритмический*** ***метод*** представляет традиционный подход к созданию программного обеспечения. Основными строительными блоками являются ***процедуры***, а внимание уделяется, прежде всего, вопросам передачи управления и декомпозиции больших алгоритмов на меньшие. Ничего плохого в этом нет, если не считать того, что системы не слишком легко адаптируются. При изменении требований или увеличении размера приложения (что происходит нередко) сопровождать их становится сложнее.

**О*бъектно-ориентированный метод*** в качестве основных строительных блоков использует ***объект*** или ***класс***. В самом общем смысле объект – это сущность, обычно извлекаемая из словаря предметной области или решения, а класс является описанием множества однотипных объектов. Каждый объект обладает идентичностью (его можно поименовать или как-то по-другому отличить от прочих объектов), состоянием (обычно с объектом бывают связаны некоторые данные) и поведением (с ним можно что-то делать или он сам может что-то делать с другими объектами).

В качестве примера рассмотрим простую трехуровневую архитектуру системы, состоящую из интерфейса пользователя, программного обеспечения промежуточного слоя и базы данных. Интерфейс содержит конкретные объекты – кнопки, меню и диалоговые окна. База данных также состоит из конкретных объектов, а именно таблиц, представляющих сущности предметной области: клиентов, продукты и заказы. Программы промежуточного слоя включают такие объекты, как транзакции и алгоритмы обработки (бизнес-правила), а также более абстрактные представления сущностей предметной области (клиентов, продуктов и заказов).

Объектно-ориентированный подход к разработке программного обеспечения является сейчас преобладающим просто потому, что он продемонстрировал свою полезность при построении систем в самых разных областях любого размера и сложности. Кроме того, большинство современных языков программирования, инструментальных средств и операционных систем являются в той или иной мере объектно-ориентированными, а это дает веские основания судить о мире в терминах объектов. Объектно-ориентированные методы разработки легли в основу идеологии сборки систем из отдельных компонентов; в качестве примера можно назвать такие технологии, как JavaBeans и СОМ+.

Если принимается объектно-ориентированный взгляд на мир, то приходится отвечать на ряд вопросов.

Какая структура должна быть у хорошей объектно-ориентированной архитектуры? Какие артефакты должны быть созданы в процессе работы над проектом?

Кто должен создавать их?

И наконец, как оценить результат?

Визуализация, специфицирование, конструирование и документирование объектно-ориентированных систем – это и есть назначение языка UML.

Унифицированный язык моделирования (UML) является стандартным инструментом для создания «чертежей» программного обеспечения. С помощью UML можно визуализировать, специфицировать, конструировать и документировать артефакты программных систем.

UML можно использовать для моделирования любых систем: от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени. Это очень выразительный язык, позволяющий рассмотреть систему со всех точек зрения, имеющих отношение к ее разработке и последующему развертыванию. Несмотря на обилие выразительных возможностей, этот язык прост для понимания и использования. Изучение UML удобнее всего начать с его концептуальной модели, которая включает в себя три основных элемента: базовые строительные блоки, правила, определяющие, как эти блоки могут сочетаться между собой, и некоторые общие механизмы языка.

Несмотря на свои достоинства, UML – это всего лишь язык; он является одной из составляющих процесса разработки программного обеспечения, и не более того. Хотя UML не зависит от моделируемой реальности, лучше всего применять его, когда процесс моделирования основан на рассмотрении прецедентов использования, является итеративным и пошаговым, а сама система имеет четко выраженную архитектуру.

UML – это язык для визуализации, специфицирования, конструирования и документирования артефактов программных систем.

Язык состоит из словаря и правил, позволяющих комбинировать входящие в него слова и получать осмысленные конструкции. В *языке моделирования* словарь и правила ориентированы на концептуальное и физическое представление системы. Язык моделирования, подобный UML, является стандартным средством для составления «чертежей» программного обеспечения.

Моделирование ПО необходимо, прежде всего, для понимания программной системы, в том числе и при изучении языков ООП. При этом единственной модели никогда не бывает достаточно. Напротив, для понимания любой нетривиальной системы приходится разрабатывать большое количество взаимосвязанных моделей. В применении к программным системам это означает, что необходим язык, с помощью которого можно с различных точек зрения описать представления архитектуры системы на протяжении цикла ее разработки.

Словарь и правила такого языка, как UML, объясняют, как создавать и читать хорошо определенные модели, но ничего не сообщают о том, какие модели и в каких *случаях* нужно создавать. Это задача всего процесса разработки программного обеспечения. Хорошо организованный процесс должен подсказать вам, какие требуются артефакты, какие ресурсы необходимы для их создания, как можно использовать эти артефакты, чтобы оценить выполненную работу и управлять проектом в целом.

С точки зрения большинства программистов, размышления по поводу реализации проекта почти эквивалентны написанию для него кода. Вы думаете – значит, вы кодируете. И действительно, некоторые вещи лучше всего выражаются непосредственно в коде на каком-либо языке программирования, поскольку код программы – это самый простой и короткий путь для записи алгоритмов и выражений.

Но даже в таких случаях программист занимается моделированием, хотя и неформально. Он может, допустим, записать набросок идеи на доске или на салфетке. Однако такой подход чреват неприятностями. Во-первых, обмен мнениями по поводу концептуальной модели возможен только тогда, когда все участники дискуссии говорят на одном языке. Как правило, при разработке проектов компаниям приходится изобретать собственные языки, и новичку непросто догадаться, о чем идет речь. Во-вторых, нельзя получить представление об определенных аспектах программных систем без модели, выходящей за границы текстового языка программирования. Так, назначение иерархии классов можно, конечно, понять, если внимательно изучить код каждого класса, но воспринять всю структуру сразу и целиком не получится. Аналогично изучение кода системы не позволит составить целостное представление о физическом распределении и возможных миграциях объектов в Web-приложении. В-третьих, если автор кода никогда не воплощал в явной форме задуманные им модели, эта информация будет навсегда утрачена, если он сменит место работы. В лучшем случае ее можно будет лишь частично воссоздать исходя из реализации.

Использование UML позволяет решить третью проблему: явная модель облегчает общение.

Некоторые особенности системы лучше всего моделировать в виде текста, другие - графически. На самом деле во всех интересных системах существуют структуры, которые невозможно представить с помощью одного лишь языка программирования. UML – графический язык, что позволяет решить вторую из обозначенных проблем.

UML - это не просто набор графических символов. За каждым из них стоит хорошо определенная семантика. Это значит, что модель, написанная одним разработчиком, может быть однозначно интерпретирована другим – или даже инструментальной программой. Так решается первая из перечисленных выше проблем.

В данном контексте ***специфицирование*** означает построение точных, недвусмысленных и полных моделей. UML позволяет специфицировать все существенные решения, касающиеся анализа, проектирования и реализации, которые должны приниматься в процессе разработки и развертывания системы программного обеспечения.

UML не является языком визуального программирования, но модели, созданные с его помощью, могут быть непосредственно переведены на различные языки программирования при наличии соответствующих программных средств. Иными словами, UML-модель можно отобразить на такие языки, как Java, C++, VisualBasic, C# и даже на таблицы реляционной базы данных или устойчивые объекты объектно-ориентированной базы данных.

Такое отображение модели на язык программирования позволяет осуществлять прямое проектирование: генерацию кода из модели UML в какой-то конкретный язык. Можно решить и обратную задачу: реконструировать модель по имеющейся реализации. Обратное проектирование не представляет собой ничего необычного. Если вы не закодировали информацию в реализации, то эта информация теряется при прямом переходе от моделей к коду. Поэтому для обратного проектирования необходимы как инструментальные средства, так и вмешательство человека. Сочетание прямой генерации кода и обратного проектирования позволяет работать как в графическом, так и в текстовом представлении, если инструментальные программы обеспечивают согласованность между обоими представлениями.

Помимо прямого отображения в языки программирования UML в силу своей выразительности и однозначности позволяет непосредственно исполнять модели, имитировать поведение систем и контролировать действующие системы.

Компания, выпускающая программные средства, помимо исполняемого кода производит и другие артефакты, в том числе следующие:

* требования к системе;
* архитектуру;
* проект;
* исходный код;
* проектные планы;
* тесты;
* прототипы;
* версии, и др.

В зависимости от принятой методики разработки выполнение одних работ производится более формально, чем других. Упомянутые артефакты – это не просто поставляемые составные части проекта; они необходимы для управления, для оценки результата, а также в качестве средства общения между членами коллектива во время разработки системы и после ее развертывания.

UML позволяет решить проблему документирования системной архитектуры и всех ее деталей, предлагает язык для формулирования требований к системе и определения тестов и, наконец, предоставляет средства для моделирования работ на этапе планирования проекта и управления версиями.

Сфера применения UML не ограничивается моделированием программного обеспечения. Его выразительность позволяет моделировать, скажем, документооборот в юридических системах, структуру и функционирование системы обслуживания пациентов в больницах, осуществлять проектирование аппаратных средств.

Для понимания UML необходимо усвоить его концептуальную модель, которая включает в себя три составные части: основные строительные блоки языка, правила их сочетания и некоторые общие для всего языка механизмы. Усвоив эти элементы, вы сумеете читать модели на UML и самостоятельно создавать их - вначале, конечно, не очень сложные. По мере приобретения опыта в работе с языком можно научиться пользоваться и более развитыми его возможностями.

Словарь языка UML включает три вида строительных блоков:

* сущности;
* отношения;
* диаграммы.

*Сущности* - это абстракции, являющиеся основными элементами модели. Отношения связывают различные сущности; диаграммы группируют представляющие интерес совокупности сущностей.

В UML имеется четыре типа сущностей:

* структурные;
* поведенческие;
* группирующие;
* аннотационные.

Сущности являются основными объектно-ориентированными блоками языка. С их помощью можно создавать корректные модели.

Классы могут существовать на нескольких уровнях представления: концептуальном, уровне анализа, уровне проектирования и уровне реализации. При моделировании объектов, выражающих концепции реального мира, очень важно отразить их действительное состояние, поведение и взаимо­отношения. Концепции, относящиеся к реализации (сокрытие информации, производительность, видимость и методы), не являются концепциями реального мира, поэтому многие потенциальные свойства класса являются на этом уровне ненужными.

*На* ***концептуальном уровне*** классы описывается в терминах реальных или предполагаемых сущностей из предметной области, а также отношения между ними.

*На* ***уровне анализа*** классы представляет собой логическую концепцию предметной области или самой программы. Аналитическая модель вообще содержит только самые общие представления о моделируемой системе. На этом уровне важно уловить основную логику системы, а не вни­кать в детали ее работы и построения.

*На* ***начальном уровне проектирования*** для классов становятся актуальны такие концепции, как сопоставление состояния с классами, эффективность переходов между объектами, отделение внеш­нее представление от внутренней реализации и определение точных сигнатур операций.

В дальнейшем классы объединяет в единый пакет информацию о состоянии и операциях, отражает основные проектные решения по локализации информации и распределению функциональности. На этом уровне классы обладает содержанием, относящимся как к реальному миру, так и непосредственно к моделируемой системе.

И*,* наконец*,* ***при реализации в программном коде*** классы принимает форму, максимально подходящую для отображения в выбранном языке программирования. При этом могут пропасть те об­щие свойства класса, которые не находят в этом языке прямого применения. На уровне реализации класс уже в точности отражает программный код.

Одна и та же модель может содержать классы нескольких уровней. При этом классы, ориентированные на непосредственную реализацию в программном коде, могут реализовывать функции логических классов данной модели. Класс программной реализации представляет собой объявле­ние класса в конкретном языке программирования и имеет ту форму, которую требует данный язык. Впрочем, довольно часто всю аналитическую информацию вместе с информацией, касаю­щейся проектирования и реализации, можно разместить в одном классе.

***Архитектура ПО*** – набор ключевых правил, определяющих организацию системы:

* совокупность структурных элементов системы и связей между ними;
* поведение элементов системы в процессе их взаимодействия;
* иерархию подсистем, объединяющих структурные элементы.

Архитектура ПО многомерна, поскольку различные специалисты работают с её различными аспектами. Различные представления архитектуры служат различным целям (модель «4+1»):

* представление функциональных возможностей ПО (представление вариантов использования);
* представление логической организации ПО (логическое представление);
* представление физической структуры программных компонент, входящих в состав ПО (представление реализации);
* представление структуры потоков управления и аспектов параллельной работы ПО (представление процессов);
* описание физического размещения компонент ПО по узлам вычислительной системы (представление размещения).



Каждое ***архитектурное представление*** – это модель системы с определенной точки зрения, в которой отражены лишь существенные аспекты и опущено все, что несущественно при данном взгляде на систему.

***Модель ПО*** – это формализованное описание системы ПО на определенном уровне абстракции. Каждая модель описывает конкретный аспект системы, использует набор диаграмм или формальных описаний и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами. Модели служат полезным инструментом анализа проблем, обмена информацией между всеми заинтересованными сторонами, проектирования ПО. Моделирование способствует более полному усвоению требований, улучшению качества системы и повышению степени ее управляемости.