Доклад

На тему:

**"Класс и объект"**

**Объектно-ориентированная декомпозиция.**

Предписывает разделение модели предметной области на элементы, выбирая в качестве критерия декомпозиции принадлежность этих элементов к различным абстракциям данной предметной области. Прежде чем разделять задачу на шаги, необходимо определить объекты предметной области.  предметная область представлена 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.

Опыт показывает, что для сложных программных систем **полезнее начинать с объектной декомпозиции,**которая имеет несколько важных преимуществ перед алгоритмической декомпозицией. Объектная декомпозиция помогает уменьшить размер программных систем за счет повторного использования общих механизмов. Объектно-ориентированные системы более гибки и проще эволюционируют со временем. 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. А уже после создания объектной модели и описания структуры каждого класса для разработки операций класса можно использовать **алгоритмическую декомпозицию для проектирования процедур.**

Врезультате **объектно-ориентированной декомпозиции**разработчик создает **объектно-ориентированные модели,**т. е. такие модели, которые фокусируют внимание на объектах заданной предметной области и решаемой задачи.

**Отличие структурной декомпозиции от объектно-ориентированной.**

Принципиальное различие между структурным и объектно-ориентированным подходом заключается в способе декомпозиции системы. Объектно-ориентированный подход использует объектную декомпозицию, при этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира. Понятие "объект" впервые было использовано около 30 лет назад в технических средствах при попытках отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программных абстракций и низким уровнем абстрагирования на уровне компьютеров. С объектно-ориентированной архитектурой также тесно связаны объектно-ориентированные операционные системы. Однако наиболее значительный вклад в объектный подход был внесен объектными и объектно-ориентированными языками программирования: Simula, Smalltalk, C++, Object Pascal. На объектный подход оказали влияние также развивавшиеся достаточно независимо методы моделирования баз данных, в особенности подход "сущность-связь".

Концептуальной основой объектно-ориентированного подхода является объектная модель. Основными се элементами являются:

• абстрагирование (abstraction);

• инкапсуляция (encapsulation);

• модульность (modularity);

• иерархия (hierarchy).

Кроме основных имеются еще три дополнительных элемента, не являющихся в отличие от основных строго обязательными:

• типизация (typing)',

• параллелизм (concurrency)',

• устойчивость (persistence).

Абстрагирование — это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Инкапсуляция — это процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать интерфейс объекта, отражающий его внешнее поведение, от внутренней реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только методы самого класса, скрыты от внешней среды. Абстрагирование и инкапсуляция являются взаимодополняющими операциями: абстрагирование фокусирует внимание на внешних особенностях объекта, а инкапсуляция (или, иначе, ограничение доступа) не позволяет объектам-пользователям различать внутреннее устройство объекта.

**Понятие класса и объекта.**

Объект — это экземпляр какого-то класса. Класс — это шаблон, в котором описаны все свойства будущего объекта и его методы. При этом если класс воздушного шарика определяет свойство цвет, то сам класс никакого значения цвета не имеет. Но экземпляры этого класса, которых, к слову, можно создавать сколько угодно, уже будут раскрашены в любые цвета.

Классы могут выстраиваться в хитрые витиеватые структуры. Чем структура хитрее, тем программа гибче, легче поддается изменениям и внедрениям нового функционала, но не обязательно. Такие слова как наследование, полиморфизм, инкапсуляция позволяют создавать структуры объектов еще витиеватее, при этом избавляют код от дублирования и делают его интуитивно понятным, но не всегда.

**Отличие объекта от класса.**

**Объект — это экземпляр класса**. Однако термин object относится к реально существующему экземпляру класса. Каждый объект должен принадлежать классу. Объекты создаются и уничтожаются в программе по мере необходимости, поэтому их время жизни ограничено. Пока объекты "живые", их свойства могут значительно меняться.

Конкретный пример поможет разобраться в том, о чем тут идет речь. Предположим, что у нас есть класс Animal (животное). Все животные имеют тела и мозги - и поэтому они могли бы быть атрибутами нашего вымышленного класса Animal. Также мы можем добавить к классу некоторые методы, которые общие у всех животных - такие как movement (перемещение), потому что все животные могут двигаться. Идея, которую Вы хотите осуществить в своем представлении, состоит в том, что этот очень общий шаблон Animal не изменяется, он просто дает несколько строк кода, которые определяют класс Animal.

Экземпляр класса Animal был бы определенным животным - львом, кошкой или зеброй. Эти экземпляры класса называются объектами. Принимая во внимание, что класс Animal - общее понятие, экземпляры этого класса - львы, коты и т. д. - берут за основу общее понятие класса и создают его реальные экземпляры.

Именно по этой причине программисты определяют конструкторы в своих классах - так происходит всегда, когда они хотят создавать объект с определенными свойствами. Например, какого вида животное должно быть, как его зовут, сколько оно весит и т. д. Так что Вы можете думать о конструкторе объекта как о чем-то таком, которое вводит класс в жизнь программы - отсюда и пошло название constructor, потому что функция конструктора создает определенный экземпляр класса.

**У объектов есть понятие "время жизни", а у класса нет**. И как ясно показывает наш пример класса Animal, каждый объект в программе имеет связанный с ним интервал времени жизни - кот или зебра не могут жить вечно. И свойства этих объектов могут меняться в течении их "жизни"; если мы определим переменную size в классе Animal, то она может обозначать размер животного (например, размер кота как объекта со временем может расти).

**Общее отличие объекта от класса**. Можно сказать, что, в то время как класс является общей концепцией предметного понятия (наподобие Animal), объект — это очень определенное воплощение этого класса, причем с ограниченным временем жизни в программе. Другой способ думать об отличии между классом и объектом - класс предоставляет шаблон для чего-то более специального, которое должен определить программист при создании объекта этого класса.

**Отношения между классами.**

Возможны следующие связи между классами в рамках объектной модели:

* Агрегация (Aggregation)
* Ассоциация (Association)
* Наследование (Inheritance)
* Метаклассы (Metaclass)

**Агрегация**

Отношение между классами типа "содержит" или "состоит из" называется агрегацией или включением. Например, если аквариум наполнен водой и в нем плавают рыбки, то можно сказать, что аквариум агрегирует в себе воду и рыбок.

Объект класса Aquarium содержит несколько объектов Fish. В то же время каждая рыбка "знает", в каком именно аквариуме она живет. Факт участия класса в отношении изображается посредством роли.

В примере можно видеть роль "home" класса Aquarium (аквариум является домом для рыбок), а также роль "inhabitants" класса Fish (рыбки являются обитателями аквариума).

Название роли обычно совпадает с названием соответствующего поля в классе.

### Ассоциация

Если объекты одного класса ссылаются на один или более объектов другого класса, но ни в ту, ни в другую сторону отношение между объектами не носит характера "владения" или контейнеризации, то такое отношение называют ассоциацией (association).

В качестве примера можно рассмотреть программиста и его компьютер. Между этими двумя объектами нет агрегации, но существует четкая взаимосвязь. Так, всегда можно установить за какими компьютерами работает какой-либо программист, а также какие люди пользуются отдельно взятым компьютером. В рассмотренном примере имеет место ассоциация многие-ко-многим.

В данном случае между экземплярами классов Programmer и Computer в обе стороны используется отношение "0..n", т.к. программист теоретически может не пользоваться компьютером (если он теоретик или на пенсии). В свою очередь компьютер может никем не использоваться (если он новый и еще не установлен).

**Наследование**

Наследование вводит иерархию "общее/частное", в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение. В качестве примера можно рассмотреть задачу, в которой необходимо реализовать классы "Легковой автомобиль" и "Грузовой автомобиль". Очевидно, эти два класса имеют много общей функциональности. Так, оба они имеют 4 колеса, двигатель, могут перемещаться и т.д. Всеми этими свойствами обладает любой автомобиль, не зависимо от того грузовой он или легковой, 5- или 12-местный. Разумно вынести эти общие свойства и функциональность в отдельный класс, например "Автомобиль", и наследовать от него классы "Легковой автомобиль" и "Грузовой автомобиль" чтобы избежать повторного написания одного и того же кода в разных классах.

Использование наследования способствует уменьшению количества кода, написанного для описания схожих сущностей, а так же способствует написанию более эффективного и гибкого кода.