Объектно-ориентированное программирование

Лекция №7. Методология объектно-ориентированного программирования

# Темы:

1. Программные продукты. Сложность ПО
2. Декомпозиция. ОО Анализ, ОО Проектирование, ОО Программирование.
3. Абстрагирование. Инкапсуляция. Модульность. Иерархия

# Простые и сложные программные системы

Мы знаем, что не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Мы не хотим сказать, что все такие системы плохо сделаны или, тем более, усомниться в квалификации их создателей. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их лучше заменить новыми, чем пытаться повторно использовать, переделывать или расширять. Разработка подобных программ скорее утомительна, чем сложна, так что изучение этого процесса нас не интересует.

Нас интересует разработка того, что мы будем называть промышленными программными продуктами. Они применяются для решения самых разных задач, таких, например, как системы с обратной связью, которые управляют или сами управляются событиями физического мира и для которых ресурсы времени и памяти ограничены; задачи поддержания целостности информации объемом в сотни тысяч записей при параллельном доступе к ней с обновлениями и запросами; системы управления и контроля за реальными процессами (например, диспетчеризация воздушного или железнодорожного транспорта). Системы подобного типа обычно имеют большое время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. В мире промышленных программ мы также встречаем среды разработки, которые упрощают создание приложений в конкретных областях, и программы, которые имитируют определенные стороны человеческого интеллекта.

Существенная черта промышленной программы - уровень сложности: один разработчик практически не в состоянии охватить все аспекты такой системы. Грубо говоря, сложность промышленных программ превышает возможности человеческого интеллекта. Увы, но сложность, о которой мы говорим, по-видимому, присуща всем большим программных системам. Говоря "присуща", мы имеем в виду, что эта сложность здесь неизбежна: с ней можно справиться, но избавиться от нее нельзя.

Сложность вызывается четырьмя основными причинами:

* сложностью реальной предметной области, из которой исходит заказ на разработку;
* трудностью управления процессом разработки;
* необходимостью обеспечить достаточную гибкость программы;
* неудовлетворительными способами описания поведения больших дискретных систем.

## Сложность реального мира.

Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований. Рассмотрим необходимые характеристики электронной системы многомоторного самолета, сотовой телефонной коммутаторной системы и робота. Достаточно трудно понять, даже в общих чертах, как работает каждая такая система. Теперь прибавьте к этому дополнительные требования (часто не формулируемые явно), такие как удобство, производительность, стоимость, выживаемость и надежность!

Эта внешняя сложность обычно возникает из-за "нестыковки" между пользователями системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, мы с трудом можем однозначно зафиксировать все его требования. Обычно они отражены на многих страницах текста, "разбавленных" немногими рисунками. Такие документы трудно поддаются пониманию, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.

Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов - схем, прототипов, - и использование системы после того, как она разработана и установлена, заставляют пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

## Трудности управления процессом разработки.

Основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса. Размер исходных текстов программной системы отнюдь не входит в число ее главных достоинств, поэтому мы стараемся делать исходные тексты более компактными, изобретая хитроумные и мощные методы, а также используя среды разработки уже существующих проектов и программ. Однако новые требования для каждой новой системы неизбежны, а они приводят к необходимости либо создавать много программ "с нуля", либо пытаться по-новому использовать существующие. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Даже если мы правильно разложим ее на составные части, мы все равно получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы она ни была, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

## Гибкость программного обеспечения.

Домостроительная компания обычно не имеет собственного лесхоза, который бы ей поставлял лес для пиломатериалов; совершенно необычно, чтобы монтажная фирма соорудила свой завод для изготовления стальных балок под будущее здание. Однако в программной индустрии такая практика - дело обычное. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость чрезвычайно соблазнительна. Она заставляет разработчика создавать своими силами все базовые строительные блоки будущей конструкции, из которых составляются элементы более высоких уровней абстракции. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. Поэтому программные разработки остаются очень трудоемким делом.

## Проблема описания поведения больших дискретных систем.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. Д. Парнас [4] пишет: "когда мы говорим, что система описывается непрерывной функцией, мы имеем ввиду, что в ней нет скрытых сюрпризов. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных". С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число в соответствии с правилами комбинаторики очень велико. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Представим себе пассажирский самолет, в котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром, сидящим на месте 38J, индивидуального освещения самолет немедленно вошел бы в глубокое пике. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что за исключением самых тривиальных случаев, всеобъемлющее тестирование таких программ провести невозможно. И пока у нас нет ни математических инструментов, ни интеллектуальных возможностей для полного моделирования поведения больших дискретных систем, мы должны удовлетвориться разумным уровнем уверенности в их правильности.

## Примеры сложных систем

### Структура персонального компьютера.

Персональный компьютер (ПК) - прибор умеренной сложности. Большинство ПК состоит из одних и тех же основных элементов: системной платы, монитора, клавиатуры и устройства внешней памяти какого-либо типа (гибкого или жесткого диска). Мы можем взять любую из этих частей и разложить ее в свою очередь на составляющие. Системная плата, например, содержит оперативную память, центральный процессор (ЦП) и шину, к которой подключены периферийные устройства. Каждую из этих частей можно также разложить на составляющие: ЦП состоит из регистров и схем управления, которые сами состоят из еще более простых деталей: диодов, транзисторов и т.д.

Это пример сложной иерархической системы. Персональный компьютер нормально работает благодаря четкому совместному функционированию всех его составных частей. Вместе эти части образуют логическое целое. Мы можем понять, как работает компьютер, только потому, что можем рассматривать отдельно каждую его составляющую. Таким образом, можно изучать устройства монитора и жесткого диска независимо друг от друга. Аналогично можно изучать арифметическую часть ЦП, не рассматривая при этом подсистему памяти.

Дело не только в том, что сложная система ПК иерархична, но в том, что уровни этой иерархии представляют различные уровни абстракции, причем один надстроен над другим и каждый может быть рассмотрен (понят) отдельно. На каждом уровне абстракции мы находим набор устройств, которые совместно обеспечивают некоторые функции более высокого уровня, и выбираем уровень абстракции, исходя из наших специфических потребностей. Например, пытаясь исследовать проблему синхронизации обращений к памяти, можно оставаться на уровне логических элементов компьютера, но этот уровень абстракции не подходит при поиске ошибки в прикладной программе, работающей с электронными таблицами.

### Структура растений и животных.

Ботаник пытается понять сходство и различия растений, изучая их морфологию, то есть форму и структуру. Растения - это сложные многоклеточные организмы. В результате совместной деятельности различных органов растений происходят такие сложные типы поведения, как фотосинтез и всасывание влаги.

Растение состоит из трех основных частей: корни, стебли и листья. Каждая из них имеет свою особую структуру. Корень, например, состоит из корневых отростков, корневых волосков, верхушки корня и т.д. Рассматривая срез листа, мы видим его эпидермис, мезофилл и сосудистую ткань. Каждая из этих структур, в свою очередь, представляет собой набор клеток. Внутри каждой клетки можно выделить следующий уровень, который включает хлоропласт, ядро и т.д. Так же, как у компьютера, части растения образуют иерархию, каждый уровень которой обладает собственной независимой сложностью.

Все части на одном уровне абстракции взаимодействуют вполне определенным образом. Например, на высшем уровне абстракции, корни отвечают за поглощение из почвы воды и минеральных веществ. Корни взаимодействуют со стеблями, которые передают эти вещества листьям. Листья в свою очередь используют воду и минеральные вещества, доставляемые стеблями, и производят при помощи фотосинтеза необходимые элементы.

# Роль декомпозиции

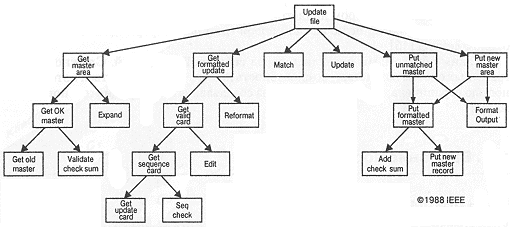
Как отмечает Дейкстра, "Способ управления сложными системами был известен еще в древности - divide et impera (разделяй и властвуй)" [16]. При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае мы не превысим пропускной способности человеческого мозга: для понимания любого уровня системы нам необходимо одновременно держать в уме информацию лишь о немногих ее частях (отнюдь не о всех). В самом деле, как заметил Парнас, декомпозиция вызвана сложностью программирования системы, поскольку именно эта сложность вынуждает делить пространство состояний системы [17].

## Алгоритмическая декомпозиция.

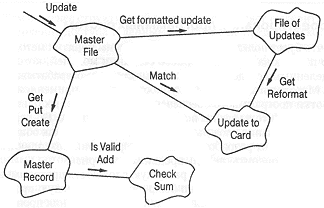
Большинство из нас формально обучено структурному проектированию "сверху вниз", и мы воспринимаем декомпозицию как обычное разделение алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса. На рис. 1 приведен в качестве примера один из продуктов структурного проектирования: структурная схема, которая показывает связи между различными функциональными элементами системы. Данная структурная схема иллюстрирует часть программной схемы, изменяющей содержание управляющего файла. Она была автоматически получена из диаграммы потока данных специальной экспертной системой, которой известны правила структурного проектирования [18].

## Объектно-ориентированная декомпозиция.

Предположим, что у этой задачи существует альтернативный способ декомпозиции. На рис. 2 мы разделили систему, выбрав в качестве критерия декомпозиции принадлежность ее элементов к различным абстракциям данной проблемной области. Прежде чем разделять задачу на шаги типа Get formatted update (Получить изменения в отформатированном виде) и Add check sum (Прибавить к контрольной сумме), мы должны определить такие объекты как Master File (Основной файл) и Check Sum (Контрольная сумма), которые заимствуются из словаря предметной области.

  
Рисунок 1: Алгоритмическая декомпозиция.

Хотя обе схемы решают одну и ту же задачу, но они делают это разными способами. Во второй декомпозиции мир представлен совокупностью автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. Get formatted update (Получить изменения в отформатированном виде) больше не присутствует в качестве независимого алгоритма; это действие существует теперь как операция над объектом File of Updates (Файл изменений). Эта операция создает другой объект - Update to Card (Изменения в карте). Таким образом, каждый объект 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. Объекты что-то делают, и мы можем, послав им сообщение, попросить их выполнить то-то и то-то. Так как наша декомпозиция основана на объектах, а не на алгоритмах, мы называем ее объектно-ориентированной декомпозицией.

  
Рисунок 2: Объектно-ориентированная декомпозиция.

Декомпозиция: алгоритмическая или объектно-ориентированная? Какая декомпозиция сложной системы правильнее - по алгоритмам или по объектам? В этом вопросе есть подвох, и правильный ответ на него: важны оба аспекта. Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия. Однако мы не можем сконструировать сложную систему одновременно двумя способами, тем более, что эти способы по сути ортогональны. Опыт показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться с приданием организованности сложности программных систем. Выше этот объектный подход помог нам при описании таких непохожих систем, как компьютеры, растения, галактики и общественные институты.

# Объектно-ориентированное программирование.

Что же такое объектно-ориентированное программирование (object-oriented programming, OOP)? Мы определяем 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. OOP 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 OOP, а называется программированием на основе абстрактных типов данных.

# Объектно-ориентированное проектирование.

Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование, напротив, основное внимание уделяет правильному и эффективному структурированию сложных систем. Мы определяем объектно-ориентированное проектирование следующим образом:

Объектно-ориентированное проектирование - это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части: объектно-ориентированное проектирование

1. основывается на объектно-ориентированной декомпозиции;
2. использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором - алгоритмами. Иногда мы будем использовать аббревиатуру OOD, object-oriented design, для обозначения метода объектно-ориентированного проектирования, изложенного в этой книге.

# Объектно-ориентированный анализ.

Объектно-ориентированный анализ (или OOA, object-oriented analysis) направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ - это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Как соотносятся ООА, OOD и OOP? На результатах ООА формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии OOP.

# Абстрагирование

## Смысл абстрагирования.

Абстрагирование является одним из основных методов, используемых для решения сложных задач. 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.

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. Мы считаем полезным еще один дополнительный принцип, называемый принципом наименьшего удивления, согласно которому абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

# Инкапсуляция

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта и реализация его методов.

Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями. Возьмем для примера структуру растения: чтобы понять на верхнем уровне действие фотосинтеза, вполне допустимо игнорировать такие подробности, как функции корней растения или химию клеточных стенок. Аналогичным образом при проектировании базы данных принято писать программы так, чтобы они не зависели от физического представления данных; вместо этого сосредотачиваются на схеме, отражающей логическое строение данных [52]. В обоих случаях объекты защищены от деталей реализации объектов более низкого уровня.

# Модульность

Понятие модульности. Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность... Однако гораздо важнее тот факт, что внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом. Классы и объекты составляют логическую структуру системы, они помещаются в модули, образующие физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Абсолютно прав Зельковиц, утверждая: "поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной" [59].

Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Такая же ситуация возникает у проектировщиков бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных схем типа НЕ, И-НЕ, ИЛИ-НЕ, но можно объединить такие схемы в стандартные интегральные схемы (модули), например, серий 7400, 7402 или 7404.

Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ (кроме самых тривиальных) лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям. Такой способ разбиения на модули хорош, но его можно довести до абсурда. Рассмотрим, например, задачу, которая выполняется на многопроцессорном оборудовании и требует для координации своей работы механизм передачи сообщений. В больших системах, подобных описываемым в главе 12, вполне обычным является наличие нескольких сотен и даже тысяч видов сообщений. Было бы наивным определять каждый класс сообщения в отдельном модуле. При этом не только возникает кошмар с документированием, но даже просто поиск нужных фрагментов описания становится чрезвычайно труден для пользователя. При внесении в проект изменений потребуется модифицировать и перекомпилировать сотни модулей. Этот пример показывает, что скрытие информации имеет и обратную сторону [60]. Деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.

Модульность - это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

Свести воедино столь разноречивые требования довольно трудно, но главное уяснить: вычленение классов и объектов в проекте и организация модульной структуры - независимые действия. Процесс вычленения классов и объектов составляет часть процесса логического проектирования системы, а деление на модули - этап физического проектирования. Разумеется, иногда невозможно завершить логическое проектирование системы, не завершив физическое проектирование, и наоборот. Два этих процесса выполняются итеративно.

# Иерархия

Что такое иерархия? Абстракция - вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Определим иерархию следующим образом:

Иерархия - это упорядочение абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия "is-a") и структура объектов (иерархия "part of").

Примеры иерархии: одиночное наследование. Важным элементом объектно-ориентированных систем и основным видом иерархии "is-a" является упоминавшаяся выше концепция наследования. Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, одиночное и множественное наследование). Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. Часто подкласс достраивает или переписывает компоненты вышестоящего класса.

Семантически, наследование описывает отношение типа "is-a". Например, медведь есть млекопитающее, дом есть недвижимость и "быстрая сортировка" есть сортирующий алгоритм. Таким образом, наследование порождает иерархию "обобщение-специализация", в которой подкласс представляет собой специализированный частный случай своего суперкласса. "Лакмусовая бумажка" наследования - обратная проверка; так, если B не есть A, то B не стоит производить от A.

Примеры иерархии: агрегация. Если иерархия "is а" определяет отношение "обобщение/специализация", то отношение "part of" (часть) вводит иерархию агрегации. В иерархии классов вышестоящая абстракция является обобщением, а нижестоящая - специализацией. Поэтому мы говорим, что класс Flower находится на более высоком уровне абстракции, чем класс Plant. В иерархии "part of" класс находится на более высоком уровне абстракции, чем любой из использовавшихся при его реализации. Так класс Garden стоит на более высоком уровне, чем класс Plant.

Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

# Использование объектного подхода

В настоящее время объектно-ориентированное проектирование - единственная методология, позволяющая справиться со сложностью, присущей очень большим системам. Однако, следует заметить, что иногда применение OOD может оказаться нецелесообразным, например, из-за неподготовленности персонала или отсутствия подходящих средств разработки.