

Сложность ПО и ОО подход

Сложность программного обеспечения

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. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. Знакомство с первыми версиями системы позволяет пользователям лучше понять и отчетливее сформулировать то, что им действительно нужно. В то же время процесс разработки 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. 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. Поэтому такой объем работ потребует привлечения команды разработчиков. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга.

Гибкость программного обеспечения. 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. Кроме того, часто приходится выполнять настройку программы под индивидуальные требования конкретного пользователя и системное окружение. Поэтому программные разработки остаются очень трудоемким делом.

Проблема описания поведения больших дискретных систем. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных. С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. 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. Всеобъемлющее тестирование таких программ провести невозможно. При неблагоприятных условиях небольшое внешнее событие может привести к критической ошибке системы.

Чем сложнее система, тем легче ее полностью развалить.

Пять признаков сложной системы

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровня.

Многие сложные системы имеют почти разложимую иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже "увидеть" такие системы и их части. Скорее всего, мы можем понять лишь те системы, которые имеют иерархическую структуру. Архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов.

2. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя. Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого.

3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять "высокочастотные" взаимодействия внутри компонентов от "низкочастотной" динамики взаимодействия между компонентами.

Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть.

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.

Иными словам и, разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты, такие как клетки, или более крупные структуры, типа сосудистых систем, имеющиеся и у растений, и у животных.

5. Любая работающая сложная система является результатом развития работавшей более простой системы... Сложная система, спроектированная "с нуля", никогда не заработает. Следует начинать с работающей простой системы.

В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы. Более того, невозможно сразу правильно создать элементарные объекты: с ними надо сначала повозиться, чтобы больше узнать о реальном поведении системы, и затем уже совершенствовать их.

Разработка сложной системы

Роль декомпозиции

Как отмечает Дейкстра, "Способ управления сложными системами был известен еще в древности – divide et impera (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 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). Именно сложность системы вынуждает делить пространство состояний системы.

Для разделения можно использовать либо алгоритмическую, либо объектно-ориентированную декомпозицию. 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. Мы должны начать разделение системы либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться

рассмотреть проблему с другой точки зрения. Опыт показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться с приданием организованности сложности программных систем.

Роль абстракции

В экспериментах Миллера было установлено, что обычно человек может одновременно воспринять лишь 7 ± 2 единицы информации. Это число, по-видимому, не зависит от содержания информации. Как замечает сам Миллер: "Размер нашей памяти накладывает жесткие ограничения на количество информации, которое мы можем воспринять, обработать и запомнить. Организуя поступление входной информации одновременно по нескольким различным каналам и в виде последовательности отдельных порций, мы можем прорвать... этот информационный затор". В современной терминологии это называют разбиением или выделением абстракций. Люди развили чрезвычайно эффективную технологию преодоления сложности. Мы абстрагируемся от нее. Будучи не в состоянии полностью воссоздать сложный объект, мы просто игнорируем не слишком важные детали и, таким образом, имеем дело с обобщенной, идеализированной моделью объекта. И хотя мы по-прежнему вынуждены охватывать одновременно значительное количество информации, но благодаря абстракции мы пользуемся единицами информации существенно большего семантического объема. Это

особенно верно, когда мы рассматриваем мир с объектно-ориентированной точки зрения, поскольку объекты как абстракции реального мира представляют собой отдельные насыщенные связанные информационные единицы.

Роль иерархии

Другим способом, расширяющим информационные единицы, является организация внутри системы иерархий классов и объектов. Объектная структура важна, так как она иллюстрирует схему взаимодействия объектов друг с другом, которое осуществляется с помощью механизмов взаимодействия. Структура классов не менее важна: она определяет общность структур и поведения внутри системы. Зачем, например, изучать фотосинтез каждой клетки отдельного листа растения, когда достаточно изучить одну такую клетку, поскольку мы ожидаем, что все остальные ведут себя подобным же образом. И хотя мы рассматриваем каждый объект определенного типа как отдельный, можно предположить, что его поведение будет похоже на поведение других объектов того же типа. Классифицируя объекты по группам родственных абстракций (например, типы клеток растений в противовес клеткам животных), мы четко разделяем общие и уникальные свойства разных объектов, что помогает нам затем справляться со свойственной им сложностью.

Эволюция объектной модели

Объектно-ориентированный подход связан со следующими событиями:

- развитие методологии программирования, включая принципы модульности и скрытия данных;
- развитие языков программирования;
- прогресс в области архитектуры ЭВМ и операционных систем;
- развитие теории баз данных;
- исследования в области искусственного интеллекта;
- достижения философии и теории познания.

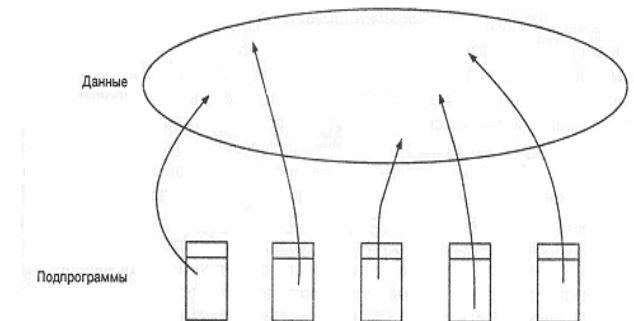
Первым, кто указал на необходимость построения систем в виде структурированных абстракций, был Дейкстра. Позднее Парнас ввел идею скрытия информации, а в 70-х годах ряд исследователей разработали механизмы абстрактных типов данных. Хоар дополнил эти подходы теорией типов и подклассов.

Развитие языков программирования:

Первое
поколение
(1954-1958)

FORTRAN I
ALGOL-58

Математические формулы

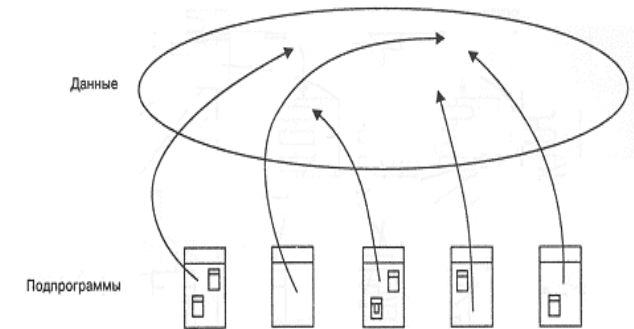


Второе
поколение
(1959-1961)

FORTRAN II
ALGOL-60
COBOL
LISP

Подпрограммы, отдельная
компиляция

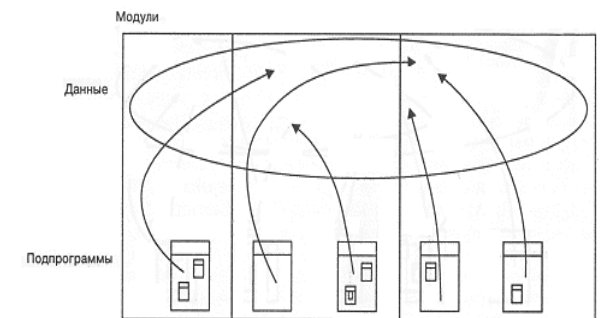
Блочная структура, типы
данных
Описание данных, работа с
файлами
Обработка списков, сборка
мусора



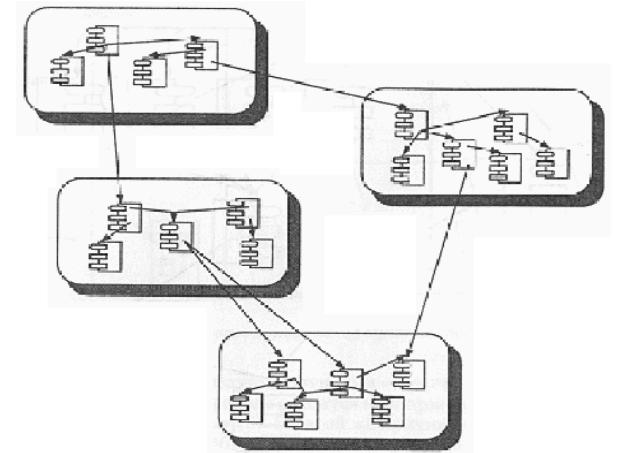
Третье
поколение
(1962-1970)

PL/I
Pascal
Simula

Объединение
возможностей
Строгая типизация, модули
Классы, абстрактные
данные



Потерянное поколение (1970-1980)	Множество языков	Проблемно-ориентированные
	Smalltalk	ООП
	Ada	Параллелизм
	C++	Шаблоны



В 80-х годах делались попытки отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программной абстракции и низким уровнем ЭВМ. По мнению сторонников этих подходов, тогда были созданы более качественные средства, обеспечивающие: лучшее выявление ошибок, большую эффективность реализации программ, сокращение набора инструкций, упрощение компиляции, снижение объема требуемой памяти. Были разработаны компьютеры Burroughs 5000, SWORD, Intel 432, IBM System/38 (AS/400). Для объектно-ориентированной архитектуры потребовались объектно-ориентированные операционные системы.

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

которой моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений.

Разработчики способов представления данных в области искусственного интеллекта также внесли свой вклад в понимание объектно-ориентированных абстракций. В 1975 г. Мински выдвинул теорию фреймов для представления реальных объектов в системах распознавания образов и естественных языков. Фреймы стали использоваться в качестве архитектурной основы в различных интеллектуальных системах.

Объектный подход известен еще издавна. Грекам принадлежит идея о том, что мир можно рассматривать в терминах как объектов, так и событий. А в XVII веке Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир. В XX веке эту тему развивала Рэнд в своей философии объективистской эпистемологии. Позднее Мински предложил модель человеческого мышления, в которой разум человека рассматривается как общность различно мыслящих агентов. Он доказывает, что только совместное действие таких агентов приводит к осмысленному поведению человека.

ОО подход

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

Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований.

Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

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

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

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.

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

На результатах ООА формируются модели, на которых основывается ООД; ООД в свою очередь создает фундамент для окончательной реализации системы с использованием методологии ООР.

Концепции ОО подхода

Каждый стиль программирования имеет свою концептуальную базу и требует своего способа восприятия решаемой задачи. Для объектно-ориентированного стиля концептуальная база – это объектная модель. Она имеет четыре главных элемента:

- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

Эти элементы являются главными в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных, имеются еще три дополнительных элемента:

- типизация;
- параллелизм;
- сохраняемость.

Называя их дополнительными, мы имеем в виду, что они полезны в объектной модели, но не обязательны.

Абстрагирование

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. Такое разделение основывается на принципе минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения и ничего больше. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу ОО проектирования.

Существует целый спектр абстракций, перечисляемых далее 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
-------------------------	---

Поведение объекта можно охарактеризовать услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. В *контрактной модели* программирования внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом. Другими словами, этот контракт определяет ответственность объекта – то поведение, за которое он отвечает.

Каждая операция, предусмотренная этим контрактом, однозначно определяется ее формальными параметрами и типом возвращаемого значения. Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется *протоколом*. Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию, полностью определяя тем самым внешнее поведение абстракции со статической и динамической точек зрения.

Центральной идеей абстракции является понятие *инварианта*. Инвариант – это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать предусловия (инварианты, предполагаемые операцией) и постусловия (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства и объект-сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять. В случае нарушения какого-либо условия обычно порождается исключительная ситуация.

Инкапсуляция

Инкапсуляция (ограничение доступа) – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части. Абстракция помогает создавать программы, а инкапсуляция упрощает их модификацию. 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.

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Практически это означает наличие двух частей в классе: интерфейса и реализации. Интерфейс отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя реализация описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части

собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

Скрытие информации – понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. "Инкапсуляция защищает от ошибок, но не от жульничества."

Модульность

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

Разделение программы на части (модули) до некоторой степени позволяет уменьшить ее сложность. 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. Выявление классов и объектов в проекте и организация модульной структуры – независимые действия, выполняемые итеративно. Модульность дополняет инкапсуляцию и абстрагирование.

В разных языках программирования модульность поддерживается по-

разному. В Smalltalk модулей нет, а в Delphi модуль – это специальная языковая конструкция.

```
unit mymodule;  
interface // интерфейс  
    function len(a,b:extended):extended;  
implementation // реализация  
    uses math; // использование модуля math  
    function len(a,b:extended):extended;  
    begin  
        len:=sqrt(a*a+b*b);  
    end;  
end.
```

В языке C++ модулями являются отдельно компилируемые файлы. Интерфейсная часть модуля помещается в заголовочный файл с расширением .h (может быть общим для нескольких модулей), а реализация помещается в файл с расширением .cpp. Заголовочный файл подключается к файлу с реализацией и в модули-клиенты с помощью директивы препроцессора #include. Такой подход строится исключительно на соглашении и не является строгим требованием самого языка.

module1.h:

```
struct point {  
    double x,y;  
    double len();  
};
```

module1.cpp:

```
#include "module1.h"  
#include <math.h>  
double point::len()  
{ return sqrt(p.x*p.x+p.y*p.y);  
}
```

module2.cpp:

```
#include "module1.h"  
...  
int main()  
{ point a;  
    cout<<a.len()<<"\n";  
    ...  
}
```

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений этот процесс можно стандартизовать, но для новых задач задача может быть очень трудной.

Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми только те элементы, которые необходимо видеть другим модулям. Конечной целью декомпозиции программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования. Структура модуля должна быть достаточно простой для восприятия; реализация каждого модуля не должна зависеть от реализации других модулей; должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны.

Если изменяется реализация, то заново компилируется только данный модуль, и программа перекомпилируется. Перекомпиляция интерфейсной части модуля, напротив, более трудоемка, так как приходится перекомпилировать все модули, связанные с данным, для очень больших

программ могут потребоваться многие часы на перекомпиляцию. Поэтому следует стремиться к тому, чтобы интерфейсная часть модулей была возможно более узкой. Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях.

При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные – за реализацию. Документирование проекта также делается по модульному принципу – модуль служит единицей описания и администрирования.

Иерархия

Иерархия – это упорядочение абстракций, расположение их по уровням.

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

Основным видом иерархии "is a" является наследование – такое отношение между классами, когда один класс заимствует структурную и функциональную часть одного или нескольких других классов (соответственно, одиночное и множественное наследование). Часто подкласс дополняет или изменяет элементы вышестоящего класса. В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля".

Наследование порождает иерархию "обобщение-специализация", в которой подкласс представляет собой специализированный частный случай своего суперкласса. Множественным наследованием часто злоупотребляют, используя его вместо агрегации. "Лакмусовая бумажка" наследования – обратная проверка; так, если В не есть А, то В не стоит производить от А. Множественное наследование усложняет реализацию языков программирования, поэтому многие языки ее не поддерживают (Delphi) или поддерживают частично (Java).

В иерархии "part of" класс находится на более высоком уровне абстракции, чем любой из использовавшихся при его реализации. Агрегация есть во всех языках, использующих структуры, 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. При уничтожении объекта главного класса должны быть уничтожены все объекты, являющиеся его частями, т.е. объект является владельцем своих составляющих.

Типизация

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

Конкретный язык программирования может иметь сильный или слабый механизм типизации, и даже не иметь вообще никакого, оставаясь объектно-ориентированным. В сильно типизированных языках нарушение согласования типов может быть обнаружено во время трансляции программы. С другой стороны, в Smalltalk типов нет: во время исполнения любое сообщение можно послать любому объекту, и если класс объекта (или его надкласс) не понимает сообщение, то генерируется сообщение об ошибке. Нарушение согласования типов может не обнаружиться во время трансляции и обычно проявляется как ошибка исполнения.

Языки, в которых типизация отсутствует, обладают большей гибкостью (например, можно создавать наборы из разнородных объектов), но даже в таких языках программисты обычно знают, какие объекты ожидаются в качестве аргументов и какие будут возвращаться. На практике при разработке сложных систем надежность языков со строгой типизацией с лихвой компенсирует некоторую потерю в гибкости по сравнению с

нетипизированными языками.

Строго типизированные языки имеют следующие преимущества:

- Все выражения будут согласованы по типу.
- Отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения.
- В большинстве систем процесс редактирование-компиляция-отладка утомителен, и раннее обнаружение ошибок просто незаменимо.
- Объявление типов улучшает документирование программ.
- Многие компиляторы генерируют более эффективный объектный код, если им явно известны типы.

По времени связывания имен объектов с их типами выделяют также статическое связывание (раннее связывание, во время компиляции) и динамическое связывание (позднее связывание, в момент выполнения программы). Концепции типизации и связывания являются независимыми, поэтому в языке программирования может использоваться разная стратегия связывания и типизации.

Одно и то же имя может означать объекты разных типов, но, имея общего предка, все они имеют и общее подмножество операций, которые можно над ними выполнять. Это особенность называется *полиморфизмом*.

Полиморфизм возникает там, где взаимодействуют наследование и динамическое связывание.

Полиморфизм наиболее целесообразен в тех случаях, когда несколько классов имеют одинаковые протоколы. Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты сами знают свой тип.

Наследование без полиморфизма возможно, но не очень полезно. Это видно на примере Ada, где можно объявлять производные типы, но из-за мономорфизма языка операции жестко задаются на стадии компиляции.

При полиморфизме связь метода и имени определяется только в процессе выполнения программ. В C++ программист имеет возможность выбирать между ранним и поздним связыванием имени с операцией. Если функция виртуальная, связывание будет поздним и, следовательно, функция полиморфна. Если нет, то связывание происходит при компиляции и ничего изменить потом нельзя.

Параллелизм

Параллелизм позволяет различным объектам действовать одновременно. Параллелизм – это способность объектов находиться либо в активном, либо в пассивном состоянии.

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс (поток управления) – это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система имеет много таких потоков: одни существуют недолго, а другие живут в течении всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

Кроме этого "аппаратного" различия, будем различать "тяжелые" процессы, для которых выделяется отдельное защищенное адресное пространство, и "легкие" процессы, которые сосуществуют в одном адресном пространстве. "Тяжелые" процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь "легких" процессов осуществляется гораздо проще, часто они используют одни и те же данные.

Многие современные операционные системы предусматривают прямую поддержку параллелизма, и это обстоятельство очень благоприятно

сказывается на возможности обеспечения параллелизма в объектно-ориентированных системах.

В то время, как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов. Объект есть понятие, на котором эти две точки зрения сходятся: каждый объект (полученный из абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется активным. Для систем, построенных на основе OOD, мир может быть представлен, как совокупность взаимодействующих объектов, часть из которых является активной.

Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с остальными объектами, действующими последовательно. Например, если два объекта посылают сообщения третьему, должен быть какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет

растерзан на части несколькими независимыми процессами.

Существует три подхода к параллелизму.

Во-первых, параллелизм – это внутреннее свойство некоторых языков программирования, таких как Ada, Smalltalk, Java и т.д. Во всех этих языках можно создавать активные объекты, код которых постоянно выполняется параллельно с другими активными объектами.

Во-вторых, можно использовать библиотеку классов, реализующих какую-нибудь разновидность "легкого" параллелизма. Ее реализация, естественно, зависит от платформы, хотя интерфейс достаточно хорошо переносим. При этом подходе механизмы параллельного выполнения не встраиваются в язык (и, значит, не влияют на системы без параллельности), но в то же время практически воспринимаются как встроенные.

Наконец, в-третьих, можно создать иллюзию многозадачности с помощью прерываний. Например, аппаратный таймер может периодически прерывать приложение и при изменении состояния внешних устройств соответствующие им объекты обращались бы, если нужно, к своим функциям вызова.

Сохраняемость

Сохраняемость – способность объекта существовать во времени, переживая породивший его процесс, и/или в пространстве, перемещаясь из своего первоначального адресного пространства.

Любой программный объект существует в памяти и живет во времени. Спектр сохраняемости объектов охватывает:

- Промежуточные результаты вычисления выражений.
- Локальные переменные в вызове процедур.
- Собственные (статические) переменные функции, глобальные переменные и динамически создаваемые данные.
- Данные, сохраняющиеся между сеансами выполнения программы.
- Данные, сохраняемые при переходе на новую версию программы.
- Данные, которые вообще переживают программу.

Традиционно, первыми тремя уровнями занимаются языки программирования, а последними – базы данных. Языки программирования, как правило, не поддерживают сохраняемость в полном объеме (исключение – язык Smalltalk). Введение сохраняемости, как нормальной составной части объектного подхода приводит к объектно-ориентированным базам данных (OODB). На практике подобные базы данных строятся на

основе проверенных временем моделей – последовательных, индексированных, иерархических, сетевых или реляционных, но программист может ввести абстракцию объектно-ориентированного интерфейса, через который запросы к базе данных и другие операции выполняются в терминах объектов, время жизни которых превосходит время жизни отдельной программы. При использовании одноуровневой памяти как в System/38 разработка OODB существенно упрощается.

Сохраняемость – это не только проблема сохранения данных. В OODB имеет смысл сохранять и классы, так, чтобы программы могли правильно интерпретировать данные. Это создает большие трудности по мере увеличения объема данных, особенно, если класс объекта вдруг потребовалось изменить.

В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако для распределенных систем желательно обеспечивать возможность перенесения объектов в пространстве, так, чтобы их можно было переносить с машины на машину и даже при необходимости изменять форму представления объекта в памяти.

Объекты

С точки зрения восприятия человеком объектом может быть:

- осязаемый и (или) видимый предмет (мяч)
- нечто, воспринимаемое мышлением (алгоритм)
- нечто, на что направлена мысль или действие (время).

С точки зрения ОО подхода объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области.

Объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве. Объект может быть определен как нечто, имеющее четко очерченные границы. Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение. Два тела, например, сфера и куб, имеют как правило нерегулярное пересечение. Хотя эта линия

пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами. Объекты могут быть осязаемыми, но иметь размытые физические границы: реки, туман или толпы людей.

Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины "экземпляр класса" и "объект" взаимозаменяемы.

Состояние

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

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер), поэтому в данном определении использован термин "обычно динамическими".

Перечень свойств объекта является, как правило, статическим, поскольку эти

свойства составляют неизменяемую основу объекта, но в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира.

Тот факт, что всякий объект имеет состояние, означает, что всякий объект занимает определенное пространство (физически или в памяти компьютера). Состояние ОО системы в целом инкапсулировано в объекты.

Поведение

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Поведение объекта определяется выполняемыми над ним операциями и его состоянием, причем некоторые операции имеют побочное действие: они изменяют состояние. Состояние объекта представляет суммарный результат его поведения.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Операция – это услуга, которую класс может предоставить своим клиентам. Выделяют следующие виды операций:

- Модификатор – операция, которая изменяет состояние объекта;
- Селектор – операция, считывающая состояние объекта, но не меняющая состояния;
- Итератор – операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности;
- Конструктор – операция создания объекта и/или его инициализации;
- Деструктор – операция, освобождающая состояние объекта и/или разрушающая сам объект.

В чисто объектно-ориентированных языках, таких как Smalltalk, операции могут быть только методами, в Delphi, C++ допускается описывать операции как независимые от объектов подпрограммы. Таким образом, можно утверждать, что все методы – операции, но не все операции – методы: некоторые из них представляют собой свободные подпрограммы.

Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет существенное значение. Это наводит на мысль представить

объект в качестве небольшого вычислительного устройства, например, конечного автомата. Объекты могут быть активными и пассивными. Активный объект имеет свой поток управления, а пассивный – нет. Активный объект в общем случае автономен, то есть он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы – источники управляющих воздействий. Если система имеет несколько потоков управления, то и активных объектов может быть несколько.

Идентичность

Идентичность – это такое свойство объекта, которое отличает его от всех других объектов.

В большинстве языков программирования для различения объектов используют имя, тем самым путая адресуемость и идентичность. В базах данных различают объекты по набору ключевых полей, тем самым смешивая идентичность и значение данных.

Объект может использоваться в программе под несколькими синонимичными именами. Такая ситуация называется структурной зависимостью и порождает в объектно-ориентированном программировании много проблем.

Трудность распознавания побочных эффектов при действиях с объектами, имеющими имена-синонимы, часто приводит к "утечкам памяти", неправильному доступу к памяти, и, хуже того, непрогнозируемому изменению состояния.

Объекты в функцию можно передавать по ссылке и по значению. По ссылке передается адрес объекта, что дает возможность изменять переданный объект, реализовать полиморфное поведение и повысить эффективность при работе со сложными объектами. При передаче по значению создается объект-копия, который имеет такое же состояние. Копирование может быть "поверхностным" (копируется только сам объект, а состояние является разделяемым) и "глубоким" (копируется объект и состояние рекурсивно). В C++ конструктор копий по умолчанию копирует объект поэлементно, что приводит к созданию синонимов на составные части (разделению состояния), когда объект содержит ссылки или указатели на другие объекты.

Аналогично, присваивание может быть сделать "глубоким" и "поверхностным".

С вопросом присваивания тесно связан вопрос равенства. Равенство можно понимать двумя способами. Во-первых, два имени могут обозначать один и тот же объект. Во-вторых, это может быть равенство состояний у двух разных объектов. В C++ нет predetermined операции равенства, поэтому нужно

самостоятельно определять операции равенства и неравенства.

Операции присваивания и равенства можно определить как виртуальные, чтобы подклассы могли переопределять их поведение.

Время жизни объектов

Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием – возвращение отведенного участка памяти системе.

Объекты могут создаваться явно или неявно. Есть два способа создать объекты явно. Во-первых, это можно сделать при объявлении – тогда объект размещается в стеке или в статической памяти. Во-вторых, можно разместить объект, то есть выделить ему память из "кучи". В С++ в любом случае при этом вызывается конструктор, который инициализирует объект. Часто объекты создаются неявно. Так, передача параметра по значению в С++ создает в стеке временную копию объекта.

Создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него. Переопределение семантики конструктора копий и операции присваивания в С++ разрешает явное управление тем, когда части объекта создаются и уничтожаются.

Уничтожение объектов также может выполняться явно и неявно. В Smalltalk и Java при потере последней ссылки на объект его забирает сборщик мусора. В языках без сборки мусора, типа C++, объекты, созданные в стеке, уничтожаются неявно при выходе из блока, в котором они были определены, но объекты, созданные в "куче" оператором new, продолжают существовать и занимать место в памяти: их необходимо явно уничтожать оператором delete. Если объект "забыть", не уничтожить, это вызовет утечку памяти. Если же объект попробуют уничтожить повторно (например, через другой указатель), последствием будет сообщение о нарушении памяти или полный крах системы.

При явном или неявном уничтожении объекта в C++ вызывается соответствующий деструктор. Его задача не только освободить память, но и решить, что делать с другими ресурсами, например, с открытыми файлами.

В некоторых системах объекты могут быть долгоживущими; под этим понимается, что их время жизни может выходить за время жизни породивших их программ. Все объекты, которым мы хотим обеспечить долгую жизнь, должны наследовать от специальных сохраняемых классов.

Отношения между объектами

Сами по себе объекты не представляют никакого интереса: только в процессе взаимодействия объектов реализуется система.

Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа иерархических соотношений объектов: связь и агрегация.

Связь – это физическое или концептуальное соединение между объектами. Объект сотрудничает с другими объектами через связи, соединяющие его с ними. Другими словами, связь – это специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому. Только вдоль связи один объект может послать сообщение другому. Связь между объектами и передача сообщений обычно односторонняя и инициализируется клиентом, но данные передаются в обоих направлениях.

Участвуя в связи, объект может выполнять одну из следующих трех функций:

- Воздействие. Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов; в определенном смысле это соответствует понятию активный объект.
- Исполнение. Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта.
- Посредничество. Такой объект может выступать как в активной, так и в пассивной роли; как правило, объект-посредник создается для выполнения операций в интересах какого-либо активного объекта или посредника.

Чтобы клиент мог послать сообщение серверу, надо, чтобы сервер был видим для клиента. В принципе есть следующие четыре способа обеспечить видимость:

- Сервер глобален по отношению к клиенту.
- Сервер (или указатель на него) передан клиенту в качестве параметра операции.
- Сервер является частью клиента.
- Сервер локально порождается клиентом в ходе выполнения какой-либо

операции.

Какой именно из этих способов выбрать – зависит от тактики проектирования.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями – это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически. Это, несомненно, отношение агрегации, но скорее концептуальное, чем физическое по своей природе.

Выбирая одно из двух – связь или агрегацию – надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом. Иногда, наоборот, предпочтительнее связи, поскольку они слабее и менее ограничительны. Принимая решение, надо взвесить все.

Объект, являющийся частью другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.

Классы

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.

Класс – это некое множество объектов, имеющих общую структуру и общее поведение.

Любой конкретный объект является экземпляром какого-то класса. Даже если класс имеет только одного представителя, этот объект не является классом, хотя в некоторых случаях класс можно рассматривать как объект.

Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но не достаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, на достаточно высоком уровне абстракции графический интерфейс пользователя, база данных или система учета как целое, это явные объекты, но не классы. Лучше считать их некими совокупностями сотрудничающих классов.

По своей природе, класс – это генеральный контракт между абстракцией и

всеми ее клиентами. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции.

Идея контрактного программирования приводит к разграничению внешнего облика, то есть интерфейса, и внутреннего устройства класса, реализации. Главное в интерфейсе – объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. Напротив, реализация класса никому, кроме него самого, не интересна. По большей части реализация состоит в определении операций, объявленных в интерфейсе класса.

Интерфейс класса обычно делится на три части:

- открытую (public) – видимую всем клиентам;
- защищенную (protected) – видимую самому классу, его подклассам и друзьям;
- закрытую (private) – видимую только самому классу и его друзьям.

Разработчик может задать права доступа к той или иной части класса, определив тем самым зону видимости клиента. Структура объекта определяется в интерфейсной части класса, а не в его реализации, чтобы

компилятор знал, сколько памяти необходимо выделить под объект. Если бы эта информация содержалась в реализации класса, нам пришлось бы написать реализацию полностью до определения его клиентов. То есть, весь смысл отделения интерфейса от реализации был бы потерян.

Классы, как и объекты, не существуют изолированно, они взаимодействуют разными способами. Между классами возникают иерархические отношения ("обобщение/специализация" и "целое/часть") и семантические, смысловые отношения, ассоциации.

Большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:

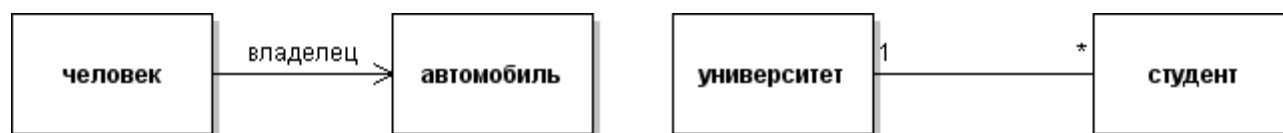
- ассоциация;
- наследование;
- агрегация;
- использование;
- инстанцирование (параметризация);
- метакласс.

Ассоциация

Из шести перечисленных видов отношений наиболее общим и неопределенным является ассоциация. Обычно аналитик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.

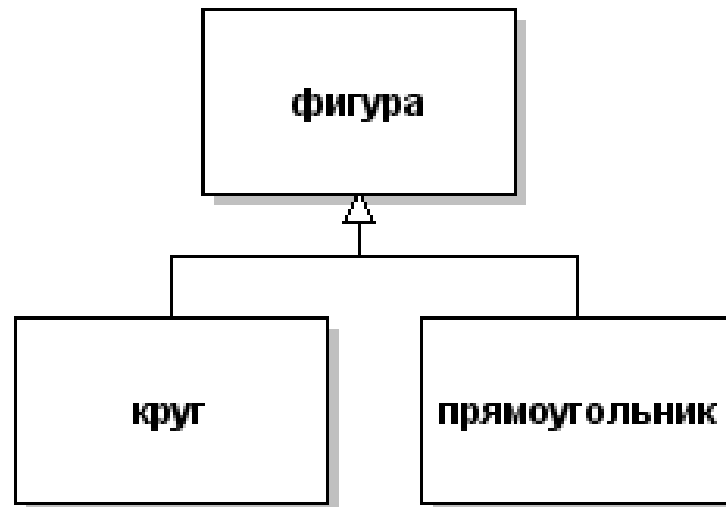
Ассоциация – смысловая связь. По умолчанию, она не имеет направления (если не оговорено противное) и не объясняет, как классы общаются друг с другом (мы можем только отметить семантическую зависимость, указав, какие роли классы играют друг для друга). Однако именно это требуется на ранней стадии анализа. Мы фиксируем участников, их роли и мощность отношения. На практике важно различать три случая мощности ассоциации:

- один-к-одному
- один-ко-многим
- многие-ко-многим



Наследование

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Обычно подклассы повторяют структуры их суперклассов. Поведение суперклассов также наследуется. В большинстве языков допускается не только наследование методов суперкласса, но также добавление новых и переопределение существующих методов. В Smalltalk любой метод суперкласса можно переопределить в подклассе. В C++ степень контроля за этим несколько выше. Функция, объявленная виртуальной, может быть в подклассе переопределена, а остальные – нет.



Самый общий класс в иерархии классов называется базовым. В многих языках программирования определен базовый класс самого верхнего уровня, который является единым суперклассом для всех остальных классов (в Smalltalk – object, в Delphi – TObject), в C++ хорошо сделанная структура классов – это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем.

У класса обычно бывает два вида клиентов:

- экземпляры;
- подклассы.

Часто полезно иметь для них разные интерфейсы. В частности, мы хотим показать только внешне видимое поведение для клиентов-экземпляров, но нам нужно открыть служебные функции и представления клиентам-подклассам. Этим объясняется наличие открытой, защищенной и закрытой частей описания класса в языке C++: разработчик может четко разделить, какие элементы класса доступны для экземпляров, а какие для подклассов.

Есть серьезные противоречия между потребностями наследования и инкапсуляции. В значительной мере наследование открывает наследующему классу некоторые секреты. На практике, чтобы понять, как работает какой-то класс, часто надо изучить все его суперклассы в их внутренних деталях.

Наследование можно рассматривать, как способ управления повторным использованием программ, то есть, как простое решение разработчика о заимствовании полезного кода. В этом случае механика наследования должна быть гибкой и легко перестраиваемой. Другая точка зрения: наследование отражает принципиальную родственность абстракций, которую невозможно отменить. В Smalltalk эти два аспекта неразделимы. С++ более гибок. В частности, при определении класса его суперкласс можно объявить public. В этом случае подкласс считается также и подтипом, то есть обязуется выполнять все обязательства суперкласса, в частности обеспечивая совместимое с суперклассом подмножество интерфейса и обладая неразличимым с точки зрения клиентов суперкласса поведением. Но если при определении класса объявить его суперкласс как private, это будет означать, что, наследуя структуру и поведение суперкласса, подкласс уже не будет его подтипом. Это означает, что открытые и защищенные члены суперкласса станут закрытыми членами подкласса, и следовательно они будут недоступны подклассам более низкого уровня. Кроме того, тот факт, что подкласс не будет подтипом, означает, что класс и суперкласс обладают несовместимыми (вообще говоря) интерфейсами с точки зрения клиента.

Одиночное наследование при всей своей полезности часто заставляет программиста выбирать между двумя равно привлекательными классами.

Это ограничивает возможность повторного использования predetermined классов и заставляет дублировать уже имеющиеся коды. Необходимость множественного наследования в ООР остается предметом горячих споров. Множественное наследование можно сравнить с парашютом: как правило, он не нужен, но, когда вдруг он понадобится, будет жаль, если его не окажется под рукой.

Проектирование структур классов со множественным наследованием – трудная задача, решаемая путем последовательных приближений. Есть две специфические для множественного наследования проблемы – как разрешить конфликты имен между суперклассами и что делать с повторным наследованием.

Конфликт имен происходит, когда в двух или более суперклассах случайно оказывается элемент (переменная или метод) с одинаковым именем. Борются с этим конфликтом тремя способами. Во-первых, можно считать конфликт имен ошибкой и отвергать его при компиляции (Smalltalk). Во-вторых, можно считать, что одинаковые имена означают одинаковый атрибут (так делает CLOS). В третьих, для устранения конфликта разрешается добавить к именам префиксы, указывающие имена классов, откуда они пришли (C++).

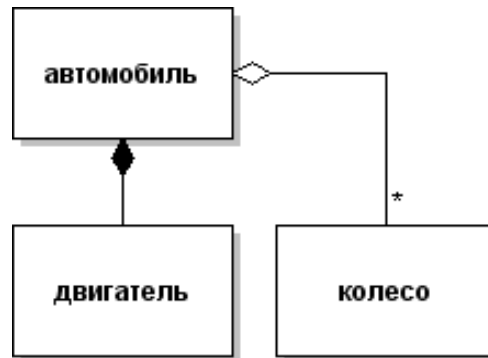
Проблема повторного наследования решается тремя способами. Во-первых, можно его запретить, отслеживая при компиляции (Smalltalk), во-вторых, можно явно развести две копии унаследованного элемента, добавляя к именам префиксы в виде имени класса-источника (C++), в-третьих, можно рассматривать множественные ссылки на один и тот же класс, как обозначающие один и тот же класс (виртуальные базовые классы C++)

При множественном наследовании часто используется прием создания примесей (mixin). Идея примесей происходит из языка Flavors: можно комбинировать (смешивать) небольшие классы, чтобы строить классы с более сложным поведением. Примесь синтаксически ничем не отличается от класса, но назначение их разное. Примесь не предназначена для порождения самостоятельно используемых экземпляров – она смешивается с другими классами, выражая какую-то одну какую-то особенность, которую можно привить другим классам через наследование. Эта особенность обычно ортогональна собственному поведению наследующего ее класса.

Агрегация

Различают физическое включение (по значению) и включение по ссылке (указателю). При использовании ссылок объекты живут отдельно друг от друга: мы можем создавать и уничтожать экземпляры классов независимо.

Чтобы избежать структурной зависимости через ссылки важно придерживаться какой-то договоренности относительно создания и уничтожения объектов, ссылки на которые могут содержаться в разных местах. Нужно, чтобы это делал кто-то один.



Агрегация является направленной, как и всякое отношение "целое/часть". Физическое вхождение одного в другое нельзя "зациклить", а вот указатели – можно (каждый из двух объектов может содержать указатель на другой).

Агрегация не требует обязательного физического включения, ни по значению, ни по ссылке. Например, акционер владеет акциями, но они не являются его физической частью. Более того, время жизни этих объектов может быть совершенно различным, хотя концептуально отношение целого и части сохраняется и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть очень косвенной. "Лакмусовая бумажка" для выявления агрегации такова: если (и только если) налицо отношение

"целое/часть" между объектами, их классы должны находиться в отношении агрегации друг с другом.

Часто агрегацию путают с множественным наследованием. Действительно, в С++ скрытое (защищенное или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Решая, с чем вы имеете дело – с наследованием или агрегацией – будьте осторожны. Если вы не уверены, что налицо отношение общего и частного (is a), вместо наследования лучше применить агрегацию или что-нибудь еще.

Использование

Отношение использования между классами соответствует равноправной связи между их экземплярами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера).



На самом деле, один класс может использовать другой по-разному. В типичном случае отношение использования проявляет себя, если в реализации какой-либо операции происходит объявление локального

объекта используемого класса.

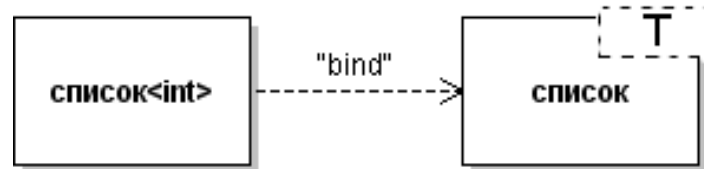
Строгое отношение использования иногда несколько ограничительно, поскольку клиент имеет доступ только к открытой части интерфейса сервера. Иногда по тактическим соображениям мы должны нарушить инкапсуляцию, для чего, собственно, и служат "дружеские" отношения классов в C++.

Инстанцирование

Существует четыре основных способа создавать параметризованные классы. Во-первых, мы можем использовать макроопределения. Например, так было сделано в раннем C++, но этот подход годился только для небольших проектов. Во-вторых, можно положиться на позднее связывание и наследование, как это делается в Smalltalk. При таком подходе мы можем строить только неоднородные контейнерные классы, так как в языке нет средства ввести нужный класс элементов контейнера; каждый элемент в контейнере трактуется как экземпляр некоторого удаленного базового класса. Третий способ реализован в языке Delphi, которые имеют и сильные типы, и наследование, но не поддерживают никакой разновидности параметризованных классов. В этом случае приходится создавать обобщенные контейнеры, как в Smalltalk, но использовать явную проверку типа объекта, прежде чем помещать его в контейнер. Наконец, есть

собственно параметризованные классы, впервые появившиеся в C++.

Параметризованный класс представляет собой что-то вроде шаблона для построения других классов; шаблон может быть параметризован другими классами, объектами или операциями. Параметризованный класс должен быть инстанцирован перед созданием экземпляров.



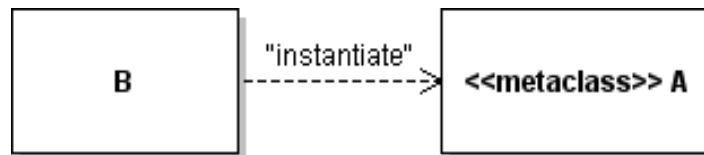
Хотя наследование является более мощным механизмом, чем обобщенные классы и через него можно получить большинство преимуществ обобщенных классов, лучше, когда языки поддерживают и то, и другое.

При проектировании обобщенные классы позволяют выразить некоторые свойства протоколов классов. Класс экспортирует операции, которые можно выполнять над его экземплярами. Наоборот, параметризующий аргумент класса служит для импорта классов и значений, предоставляющих некоторый протокол. С++ проверяет их взаимное соответствие при компиляции, когда фактически и происходит инстанцирование. Например, мы могли бы определить упорядоченную очередь объектов, отсортированных по некоторому критерию. Этот параметризованный класс должен иметь аргумент (класс Item), и требовать от этого аргумента

определенное поведение (наличие операции <). При инстанцировании в качестве класса Item годится любой класс, который имеет соответствующий протокол. Таким образом, поведение классов в семействе, происходящем от одного параметризованного класса, может изменяться в весьма широких пределах.

Метакласс

Любой объект является экземпляром какого-либо класса. Что будет, если мы попробуем и с самими классами обращаться как с объектами? Для этого нам надо ответить на вопрос, что же такое класс класса? Ответ – это метакласс. Иными словами, метакласс – это класс, экземпляры которого суть классы. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках. Главное назначение – возможность экспериментировать с другими объектно-ориентированными парадигмами и создавать такие инструменты для разработчика, как браузеры классов и объектов.



Метаклассы есть в Smalltalk и Delphi, но не в C++. Первичное назначение метакласса – поддержка переменных класса (которые являются общими для

всех экземпляров этого класса), операций инициализации переменных класса и создания единичного экземпляра метакласса. Хотя в С++ метаклассов нет, семантика его конструкторов и деструкторов служит целям, аналогичным тем, что вызвали к жизни метаклассы. С++ имеет средства поддержки и переменных класса, и операций метакласса (элементы данных или методы класса описанные как статические).

Выявление классов и выбор операций

Опыт показывает, что процесс выделения классов и объектов является последовательным, итеративным. Очень важно, следовательно, с самого начала по возможности приблизиться к правильным решениям, чтобы сократить число последующих шагов приближения к истине. Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев:

- зацепление;
- связность;
- достаточность;
- полнота;
- примитивность.

Зацепление – это степень глубины связей между отдельными модулями, классами и объектами. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать. Сложность системы может быть уменьшена путем уменьшения зацепления между отдельными модулями. Существует определенное противоречие между явлениями зацепления и наследования. С одной стороны, желательно избегать сильного зацепления классов; с другой стороны, механизм наследования, тесно связывающий подклассы с суперклассами, помогает выгодно использовать сходство абстракций.

Связность – это степень взаимодействия между элементами отдельного модуля, класса или объекта, характеристика его насыщенности. Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели.

Под достаточностью подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Иначе говоря, компоненты должны быть полностью пригодны к использованию. Для примера рассмотрим класс `set` (множество). Операция удаления элемента из

множества в этом классе, очевидно, необходима, но будет ошибкой не включить в этот класс и операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию.

Под полнотой подразумевается наличие в интерфейсной части класса всех характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все аспекты абстракции. Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все для взаимодействия с пользователями. Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, вынося на верх такие операции, которые можно реализовать на более низком уровне.

Из этого вытекает требование примитивности. Примитивными являются только такие операции, которые требуют доступа к внутренней реализации абстракции. Так, в примере с классом `set` операция добавления к множеству элемента примитивна, а операция добавления четырех элементов не будет примитивной, так как вполне эффективно реализуется через операцию добавления одного элемента. Конечно, эффективность тоже вещь субъективная. Операция, которая требует прямого доступа к структуре данных, примитивна по определению. Операция, которая может быть

описана в терминах существующих примитивных операций, но ценой значительно больших вычислительных затрат, также является кандидатом на включение в разряд примитивных (например, операция добавления к множеству другого множества).

Описание интерфейса класса или модуля – трудная работа. Обычно первое приближение делается, исходя из структурного смысла класса, а затем, когда появляются клиенты класса, интерфейс уточняется, модифицируется и дополняется. В частности может возникнуть потребность в создании новых классов или в изменении взаимодействия существующих.

В пределах каждого класса принято иметь только примитивные операции, отражающие отдельные аспекты поведения. Такие методы называются точными. Принято также отделять методы, не связанные между собой. Это облегчает образование подклассов с переопределением поведения. Решение о количестве методов может быть обусловлено двумя причинами: описание поведения в одном методе упрощает интерфейс, но усложняет и увеличивает размеры самого метода; расщепление метода усложняет интерфейс, но делает каждый из методов проще.

В объектно-ориентированном проектировании принято рассматривать методы класса как единое целое, поскольку все они взаимодействуют друг с другом для реализации протокола абстракции. Таким образом, определив

поведение, нужно решить, в каком из классов это поведение реализуется. Критериями для принятия решения служат следующие вопросы:

Повторная используемость: Будет ли это поведение полезно более чем в одном контексте?

Сложность: Насколько трудно реализовать такое поведение?

Применимость: Насколько данное поведение характерно для класса, в который мы хотим включить поведение?

Знание реализации: Надо ли для реализации данного поведения знать секреты класса?

Обычно операции объявляются как методы класса, к объектам которого относятся данные действия, но многие языки допускают описание операций в виде свободных подпрограмм.