**Волківський Віталій Васильович**

**Структура складної системи**

Із загальної теорії складних систем для сфери розробки ПЗ застосували основні принципи для опису ПЗ.

*Теорія складних Систем Месарович Акахара*

**Ознаки складних систем**

* Складні системи частіше являються ієрархічними. Складаються із взаємозалежних підсистем, які в свою чергу можуть бути розділені на підсистеми і т.д., аж до найнижчого рівня.
* Вибір, які компоненти в даній системі являються елементарними - відносно довільний, і більшій мірі лишається на погляд розробника.(нижній рівень для одного наглядача, може бути високим для іншого).
* Внутрішньо-компонентний зв’язок сильніший, ніж зв’язок між компонентами. Це дозволяє відділити високочастотну взаємодію всередині компонентів від низькочастотної динаміки між компонентами.(це дає можливість відносно ізольовано вивчати компоненти).
* Ієрархічні системи, зазвичай, складаються з небагатьох типів підсистем, по різному скомбінованих і організованих.(різні складні системи мають однакові структурні зв’язки).
* Будь-яка працююча складна система являється результатом розвитку працюючої більш простої системи. Складна система, спроектована з нуля ніколи не запрацює з першої спроби.

Канонічна форма складної програмної системи(див. зошит).

Система розглядається з двох точок зору. Будується ієрархія з двох видів: структура-клас, структура-об’єкт. Кожна з ієрархій являється багаторівневою. Класи і об’єкти більш високого рівня побудовані із комбінацій більш простих. Який клас або об’єкти вважається елементарним, обирається в залежності від поставленої задачі. Структури класів і об’єктів пов’язані між собою. Кожен елемент структури об’єкту – це деякий екземпляр деякого класу. В складній системі об’єктів, зазвичай на порядок або кілька порядків більше ніж класів. В класі зібрані загальні властивості, що описують всі екземпляри цього типу. Структуру класу і структуру об’єкту разом, часто називають архітектурою системи.

При аналізі складної програмної системи виявляється велика кількість взаємодіючих один з одним частин. В процесі проектування ПЗ необхідно одночасно слідкувати за багатьма частинами. Існують психологічні досліди, що людський мозок одночасно сприймає 7 +- одиниць інформації. Складність програмної системи, що розробляється постійно збільшується. Існує потреба в спеціальних прийомів подолання.

**Декомпозиція**

Принцип декомпозиції при проектуванні блоків був спроектований Дейксрою.

При проектуванні складної програмної системи необхідно розділяти її на менші і менші підсистеми, кожну з яких можна вдосконалювати незалежно.

**Два види декомпозиції для розробки ПЗ:**

* Алгоритмічна декомпозиція – в цьому випадку декомпозиція – це розділення алгоритмів, де кожен новий виконує один з етапів загального процесу. Алгоритмічна декомпозиція розроблялась в рамках структурного програмування С++.
* Об’єктно-орієнтована декомпозиція. В цьому випадку предметна область представляється сукупністю автономно взаємодіючих обличь, які взаємодіють один з одним, щоб забезпечити поведінку системи, що відповідає більш високому рівню. Граф – зв’язок між вершинами з’являється тоді, коли є хочаб одне повідомлення між собою. Кожна вершина в графі – це об’єкт, який має свою власну поведінку. Обмін повідомленнями – передача інформації між об’єктами і виклик функції.

Історично першою була алгоритмічна декомпозиція. Однак алгоритмічну декомпозиції використовують досі. рекомендують починати з об’єктної декомпозиції, а потім застосовувати алгоритмічну.

**Абстракція**

За допомогою абстрактції будується загальна і ідеалізована модель предметної області, при цьому недуже важливі деталі ігноруються. Тобто кількість одиниць інформації, яку потрібно одночасно аналізувати сильно зменшується.

**Ієрархія**

Всередині системи створюється впорядковані рівні. Ієрархії класів і ієрархії об’єктів. Структура класу визначає загальну структуру і поведінку всередині системи. Структура об’єктів – показує схему взаємодії об’єктів між собою. Динаміку системи – об’єкти, статику – класи.

**Об’єктна модель**

**Покоління мов програмування**

Історія розвитку програмування починається в середині 50-х років і вміщує в собі дві тенденції:

* Зміщення акцентів окремої деталі до програмування більш крупних компонентів
* Розвиток і вдосконалення мов програмування високого рівня.

Огляд мов програмування з точки зору абстракцій.

|  |  |  |
| --- | --- | --- |
| 1953-58 | Fortran  Algol-58  Flow matic  IPL V |  |
| 59-62 | Fortral II | підпрограми |
| Algol 60 | Блочна структура, типи даних |
| COBOL | Робота з файлами, опис даних |
| Lisp | Вказівники, збір сміття, обробка списку |
| 63-70 | PL/J | Fortran+algol+cobol |
| Algol 68 |  |
| Pascal | Більш строгий розвиток Алгол 60 |
| Simula | Абстрактні дані і класи |
| 70-80 | Smalltalk |  |
| C++ |  |
| Clos |  |
| Eiffel |  |
| Ada |  |
| Від 90-х | RAD, Java |  |
|  |  |
|  |  |

В кожному поколінні сильно змінювались підтримувані мовами механізми абстракції. Мови першого покоління орієнтувались на математичні розрахунки, научно інженерне застосування. Словник предметної області був виключно математичним.

В другому покоління основною тенденцією став розвиток алгоритмічних абстракцій. Потужність комп’ютерів швидко росла. Сфера застосування комп’ютерів розширювалась. Відповідно це ще один крок переходу предметної області від машини.

Продуктивність почала рости майже кспотенціально, з’явилась технічна можливість вирішувати більш важкі задачі виникла потреба в обробці найрізноманітніших типів даних. В мові Pascal I Simula з’явилася пітримка абстракції даних. Програмісти змогли описувати свої власні комбінації типів даних. Це стало ще одним кроком в сторону предметної області, його прив’язки конкретної машини.

Був шалений сплеск активності. Було створено більше 2000 мов програмування. В багатьох мовах програмування прєодалєвалось наслідування обмежень минулих мов, пов’язаних з описом предметної області в великих маштабах. Smalltalk – повністю перероблена мова символів, і це перша повністю об’єктно орієнтована мова. C++ доробка стандарту С або C + Simula. Clos – подальний розвиток мови Lisp. Eifel – розвиток Simula I Ada. Ada розвиток Алгол-68.

В рамках 5-го покоління цікавлять мови програмування в рамках об’єктно орієнтованого програмування.

**Топологія мов першого і початку другого покоління**

під топологією розуміється основні елементи мови програмування і їх взаємодія. У всіх програм область даних – спільна. Програми. Реалізовані на цих мовах мали відносно просту структуру, що складається тільки з глобальних даних і підпрограм. Дані спільні. Дані не можуть бути розділені по типах. Як наслідок в великих системах важко або нереально гарантувати цілісність даних.при експлуатації системи виникає велика кількість перехресних зв’язків.

**Топологія мов кінця другого, початку третього**

з’явилися вкладеності(підпрограми)

* Були розроблені мови, що підтримують різні способи передачі параметрів.
* В мови були вкладені принципи структурного програмування(можна описувати вкладені один в одного підпрограми)
* Виникли методи структурного проектування, в рамках яких великі системи створювались використувуючи підпрограми.

**Топологія мов кінця третього покоління**

В цей період при створенні великих програмних систем почали активно застосовувати колективний спосіб розробки. Відповіддю на цю необхідність став окремо компільований модуль. Модуль може збирати набір даних і підпрограм, які будуть використовуватися спільно. В більшості мов не було правил взаємодії.

**Топологія об’єктних і об’єктно орієнтованих мов**

Основний елемент топології – модуль, але він складається із логічно пов’язаних об’єктів і класів, а не підпрограм, як в мовах перших поколінь. Структура програм малої та середньої складності. Основними складними та логічними. Якщо програмна система дуже складна – застосовується масштабування, класи і об’єкти об’єднуються в кластери.

**Об’єктно-орієнтоване програмування** – методологія програмування, основана на представленні програми у вигляді сукупності об’єктів, кожен з яких являється екземпляром певного класу, а класи створюють ієрархію наслідування.

Мають бути три складові

1. Основа – об’єкти, а не алгоритми
2. Об’єкт – завжди екземпляр певного класу
3. Класи завжди організовані ієрархічно

Відповідно і мови програмування є об’єктно орієнтовані. Якщо немає наслідування, то ці мови називаються об’єктні(Ада).

**Об’єктно-орієнтоване проектування(OOD)** – методологія проектування, яка поєднує в собі процес, об’єктної декомпозиції, а також прийоми представлення логічної і фізичної, статичної та динамічної моделі представлення системи. Частіше представлення роблять за допомогою набору графічних діаграм(UML).

**Об’єктно-орієнтований аналіз** – методологія, при якій вимоги до системи сприймаються з точки зору класів і об’єктів, виявлених в предметній області.

Взаємозв’язок: 1 – ООА, потім проектування ООД, потім ООП

**Складові частини об’єктного підходу**

Існують поняття стилю або парадигми програмування – це спосіб побудови програм, оснований на певних принципах програмування, а також вибір підходящої мови, який робить програми, написані в цьому стилі зрозумілими.

Виділяють 5 основних різновидів стилю програмування:

1. процедурно-орієнтований стиль(парадигма)(ключова абстракція – алгоритм, орієнтований на обрахунки),
2. об’єктно-орієнтований(класи і об’єкти),
3. логіко-орієнтований стиль(цілі, часто виражені в термінах исчислениях при декатах),
4. стиль, орієнтований на правила(если - то). В рамках цієї парадигми створюються експертні системи.
5. Стиль, орієнтований на обмеження(інваріантні співвідношення)

Неможна сказати, що якийсь стиль являється краще за інших, і може застосовуватись в усіх сферах. Для розрахункових задач – процедурно-орієнтований. Для проектування баз знань – на правилах. Однак можна сказати, що Об’єктно-орієнтований стиль може застосовувать для широкого кола задач, і на ньому можна базувати інші парадигми(більш універсальні).

Об’єктно-орієнтований стиль базується на об’єктній моделі, вона складається з наступних елементв(1-4 основні):

1. Абстрагування
2. Інкапсуляція
3. Модульність
4. Ієрархія
5. Типізація
6. Паралелизм
7. Зберігаємість

**Абстрагування**

Абстракція виділяє істотні характеристики деякого об’єкту, які відрізняють його від інших об’єктів, таким чином чітко визначає його межі з точки зору розробника. Вибір правильного набору абстракцій для заданої предметної області, це і є головна задача ООПр.

**Інкапсуляція**

Поняття інкапсуляції на перших етапах розвитку об’єктно-орієнтованого підходу мало на увазі можливість об’єднання і даних і функцій обробки цих даних у вигляді одного нового типу даних – клас. Пізніше під інкапсуляцією стали розуміти об’єднання в класі двох частин: інтерфейсу і реалізації. Інтерфейс відображає зовнішню поведінку об’єкту, реалізація – внутрішнє представлення цієї абстракції, а також механізми досягнення бажаної поведінки.

Інкапсуляція – процес відділення один від одного елементів об’єктів, які визначають його влаштування і поведінку в рамках одного типу. Інкапсуляція служить для того щоб ізолювати обов’язки абстракцій…

**Модульність**

Модульність – властивість системи, яка була розкладена на внутрішньо сильно пов’язані, але слабо пов’язані між собою модулі. В модулях явно виділяють інтерфейсну частину і реалізацію.

**Ієрархія** – впорядкування абстракцій і розміщення їх по рівнях. В ієрархії класів впорядкування робиться за допомогою наслідування.

**Типізація** – спосіб захистити від використання об’єктів одного типу замість іншого. Або керувати таким використанням. Існують мови з сильною типізацією(Pascal).

Клас – це тип. Об’єкт – змінна типу клас.

**Паралелизм**

Сучасні програмні системи характеризуються паралельністю в двох аспектах: коли одна задача розбивається на кілька програмних ниток(thread), які можуть виконуватись умовно паралельно. Обробляються важливі питання по синхронізації ниток. Розпаралелювання однієї задачі на кілька процесів. Разом.

**Зберігаємість** – здатність об’єкту переміщатися із свого початкового адресного простору, і існувати в часі, переживаючи породивший його процес.

Спектр зберігаємості об’єктів охвачує(реалізуються мовами програмування(4-5 займаються бази даних)):

1. Проміжні результати обрахувань
2. Локальні змінні у виклику процедур
3. Глобальні змінні і динамічно створювані данні
4. Дані, які зберігаються між сеансами виконання програми
5. Дані, зберігаємі при переході на нову версію програми
6. Дані, які переживають рпограму.

Структури і клас

Клас – тип даних, що визначається користувачем. Екземпляр класу – об’єкт. До появи класу використовувались структури.

Струткура – користувацький тип, який може об’єднувати в собі різнотипні дані.

Приклад для представлення даних

Struct date{

Int month, day, year;

};

Date today;

Void setDate(date\*, int, int, int);

Void getDate(date\*);

При такій реалізації:

1. Мається спеціалізований тип структури для зберігання тільки даних
2. Набір функцій для обробки даних типу date

В імені функції зазвичай вказується ім’я типу, для якого вона описується. Найбільший недолік такого підходу в тому, що явних зв’язків між функціями і типом даних немає.

Пізніше в мовах програмування з’явилася можливість об’являти функції всередині структури.

Функції, об’явлені всередині структури – називаються функції члени. Вони викликаються через використання ім’я.функція.

За замовчуванням у структурах доступ – паблік.

В компіляторх С струткура за замовчуванням розглядається як звичайний клас, у якого всі доступи відкриті.

Особливості структур в C#

Struct point{

Public int x;

Public int y;

Public void increment(){x++; y++;}

Public void Display()

{

Console.writeline(“x= “ + x, “y= ” + y);

}

Int main()

{

Point myPoint;

myPoint.x = 5;

myPoint.y = 5;

myPoint.Display();

}

Навідміну від С++, струткура не являється особливим видом класу. Для структури заборонено робити наслідування. Крім того маються специфічні можливості при створенні об’єкту структури.

myPoint.x = 5;

myPoint.Display(); //помилка, бо Y не ініціалізовано.

//

Point p = new Point();

p.Display

різниця між структурами і класами в тому що об’єкти структур розміщуються стеці, а не в купі.

Класи

В мові С++ подальший розвиток структур і функцій всередині структур отримали класи

Class date{

Int day, month, year;

Public:

Void set(int,int,int);

Void get(int,int,int);

Void print();

}

Особливості:

1. Блок членів даних(деньб місб рік) знаходиться в розділі приват
2. В класі завжди існує розділ паблік(те що робить відкриту(інтерфейсну) частину)
3. Набір функцій в розділі паблік реалізує інтерфейсну частину класу
4. Додалася функція get, яка зчитує значення приватних даних
5. Всередині функцій членів доступ до даних є.

Посилання на себе

В мові С++ службове слово this використовується як вказівник на об’єкт, через який викликається функція-член. Таким чином можна вважати, що в будь якому класі існує неявне представлення типу. Без this не обійтися.

Ініціалізація об’єкту, або конструктори

В прикладі дати за рахунок розділу приват ми обеспечили себе від прямого доступу до полів даних, але залишається ризик початкової ініціалізації створюваного об’єкту. Виходом було створення конструктора, який вионується лише раз.

Є конструктори:

1. За замовчуванням
2. Ініціалізації
3. Копіювання

Особливості конструкторів С++:

1. Ім’я конструктора = ім’я класу
2. Параметри можуть бути будь-які або бути відсутніми
3. Тип результату відсутній
4. Можна перевантажувати

**Особливості класів в C#**

В класі явно відсутні розділи специфікаторів доступу. Вони вказуються перед кожним членом даних

Для створення нового об’єкту обов’язково потрібно писати new

В мові С++ nw являється оператором виділення пам’яті для вказівників, для динамічно створюваних типів застосовуються оператори new і delete.

В C# new – це ключове слово, яке розміщує створений об’єкт типу клас, і тільки після цього до членів об’єкту можна звертатися. До цього це просто ім’я, доступу до якого немає.

Оператора вивільнення пам’яті не потрібно створювати. Існує збиральник сміття.

**Особливості конструкторів в C#**

Конструктор за замовчуванням можна не визначати. В цьому ипадку всі поля даних отримають значення за замовчуванням для своїх типів. При визначенні будь-якого іншого конструктора, конструктор за замовчуванням видаляється з класу. Рекомендується при додаванні будь-якого іншого спеціального конструктора в клас, додатково явно писати код конструктора без параметрів.

Об’єднання кількох конструкторів, застосовуючи не обов’язкові аргументи

**Ключове слово this**

1. Уточнення імені об’єкту. Забеспечує доступ до теперішнього екзампляра класу
2. Застосування this при щепленні конструкторів. Якщо ми хочемо оголошувати змінні з будь-якою кількістю параметрів потрібно зробити щеплення. Описується універсальний конструктор, коли всі параметри явно передаються, а інші описуються з необхідними комбінаціями.

Public car(){}

Public car(string pn):this(pn,10){}

Public car(int cs):this(“”, cs){}

Public car(string pn, int cs){petName = pn; currSpeed = cs;}

**Ключове слово static**

Описує статичні поля і методи. Статичний метод і статичне поле є спільним для всіх об’єктів. Статичні члени можна використовувати через ім’я класу, не створюючи об’єкт.

**Перевантаження операцій**

Програми працюють з об’єктами, які являються конкретними представленнями абстрактних понять. +,-,\*,/ стандартна реалізація математичних понять цілих чисел. Такі поняття включають в себе множину операцій, які в зручній, короткій і звичній формі виконують дії. Мови програмування безпосередньо підтримують обмежену кількість таких понять. Наприклад С++за замовчуванням не підтримає комплексну арифметику, матричну алгебру і т.д. в С++ з’явилася можливість за допомогою класів представляти неелементарні об’єкти і дії над ними. Виникла потреба дій над такими неелементарними об’єктами відображати в загальноприйнятому вигляді.

Class complex{

Double re, int;

Public: complex(double r, double i){re=r; im=I;}

Friend complex operator+(complex, complex);

Friend complex operator\* (complex, complex);

Перелік операцій, які можна перевантажувати:

+-\*/%^&1`!=<>+=-=\*=/=%=^=!=<<>>>>=<<==!=⬄=&&||++--[]()newdelete

На переванаження операцій існують обмеження:

1. Неможна придумати власне позначення для нової операції
2. Неможна змінити кількість операндів в операції
3. Тернарні не перевантажуються
4. Неможна змінити приорітет операції
5. Операцію можна викликати за допомогою явного виклику

Бінарні і унарні операції

Бінарна може бути визначена або як функція член, що отримує один параметер або як дружня функція з двома параметрами

[aa.operator@(bb)](mailto:aa.operator@(bb)) = aa@bb = operator@(aa,bb)

унарна операція може бути визначена як функція член без параметрів або як дружня функція з одним параметром.

Визначені значення операцій

Відносно сенсу операцій, що визначає користувач не робиться ніяких здогадок. Наприклад для операції привласнення не робиться перевірка сенсу. Між деякими операціями існує деякий логічний зв’язок і взаємозв’язок перевантажених операцій.

Перетворення типу, що визначається.

Для кожного нового типу, з яким виконуватимуться операції, програміст має додатково визначати ще дві версії операції, при цьому код дуже схожий. І в випадку якоїсь виявленої помилки, доведеться відлагоджувати код для блоку однакових функцій.

Виходом являється написання конструктора перетворення, який допомагає зрозуміти як зовнішній, по відношенню до розроблюваного, тип трактується в нашому класі.

Complex(double r){re = r; im = 0;}

Operator(complex, complex(double));

Неоднозначності, який компілятор не може сам вирішити.

Class x{

X(int);

X(char\*);

}

Class y{

Y(int);

}

Class z{

Z(x);

}

X f(x);

Y f(y)

Z g(z);

F(1);

При виклику ф(1) компілятор повідомить про неожнозначність, в який тип перетворити. Для цього необхідно явно вказати

F(x(1));

F(y(1);

G(“ryadok”);

Два послідовних перетворення не беруться.

Для того щоб розрізняти між собою парефіксну операцію С++

X operator ++();

X operator ++(int);

В ході першої функції, змінений результат повинен повернутися в якості результату самої функції. В другій функції в коді, спочатку, зберігаються проміжні зачення, воно збільшується, але повертається старе.

Особливості операції привласнення

Мова С++ дощволяє перевантажити оператор привласнення. Якщо оператора привласнення не визначено, то в виразі а=б буде побітове копіювання. Якщо динамічних членів немає, то норм. А якщо хоч один динамічний член, то це не норм. Таким чином якщо в деякому класі Х є динмічно виділяємі члени даних, то для коректного використання в ньому має бути наступні 4 складові:

Class x{

X();

X(X&);

Operator=(X&);

~X();

}

**Наслідування класів**

Видимість в класах при наслідуванні

При наслідуванні в С++ можна явно вказувати тип видимості

|  |  |  |  |
| --- | --- | --- | --- |
| Тип наслідування/було в базовому класі | public | protected | private |
| public | public | protected | private |
| protected | rpotected | protected | private |
| private | private | private | private |

За допомогою типу наслідування видимість може бути тільки додатково захищена(посилена). За замовчуванням при наслідуванні струткур тип наслідування паблік, а для класів прайват.

Можна додатково керувати і переміщувати члени даних наступним чином:

Class manager:employee

{

Public:

Employee: :name;

Employee:department;

}

Сталося закрите наслідування. Всі члени даних, члени функції, що прийшли від емплоі стали прайват. В розділі паблік просто згадуються ті члени даних, які мають стати паблік.

**Наслідування і вказівники**

Class base{…}

Class derived:public base{…}

Derived m;

Base\* pb=&m; //неявне перетворення типів

Derived\* pd=pb; //помилка

Pd=(derived\*)pb; //явне перетворення

1. Автоматично робиться неявне перетворення типу. На об’єкт успадкованого класу може бути направлений вказівник типу базовий клас
2. Вказівник на успадкований об’єкт не можу бути направлений на об’єкт базового типу.
3. Примусове явне перетворення вказівника(об’єкта на який посилається) pb в успадкований тип.

Об’єкт похідного класу(при відкритому наслідування) при роботі з ним через вказівник можна розглядати як об’єкт його базового класу. Обернене – невірно.

Class base

{

Int m1;

Public:

Int m2;

}

Class derived:base{…};

Main()

{

Derived d;

d.m2=2;

base\* pb=&d;

pb->m2=2;

pb=(base\*)&d;

pb->m2=2;

}

Закрите наслідування. M2 став закритим членом даних всередині derived.

Далі буде помилка: немає доступу до закритого члена даних m2. Потім йде спроба отримати доступ до поля m2 через вказівник pb через базовий клас. Перетворення типу через закрите наслідування неможливе. Потім спроба явного перетворенння. Далі ми отримали доступ до закритого наслідуванням полю, яке раніше було паблік. Далі Таким чином доступ до поля m1 отримати неможна.

**Ієрархія класів**

Похідний клас може бути базовим класо для іншої цепочки класів. В загальному вигляді в С++ допускається множинне наслідування: наявність кількох предків для одного класу.

Class employee{};

Class manager^public employee{};

Class director:public manager{};

Class temporary{};

Class secretar:public employee{};

Class tsec:public secretar, public temporary{};

Class consulrant:public temporary, public manager{};

В таких випадках малюють діаграму класів(юмл). Стрілки малюються вгору.

Особливості наслідувавння в C#

Class car

{

Public readonly int maxspeed;

Private int currspeed;

Public car(int max){maxspeed=max}

Public car(){maxspeed=100;}

Pub;lic car(){

Get{return currspeed;}

Set{corrspeed=value;

If(currspeed>maxspeed)

Currspeed=maxspeed;

}

}

}

Class minivan:car{}

Main()

{

Minivan myvan=new minivan();

Myvan.speed=10;

//…

myVan.currspeed=30;// буде помилочка

}

Множинне наслідування в C# не підтримається. Але є спосіб обійти це обмеження за допомогою реалізації інтерфейсу.

Ключове слово **sealed**

Воно забороняє подальше наслідування класів. Пишеться перед словом class.

В C# можна сказати що структури неявно запаковані(неможна наслідуввати).

Ключове слово **protected**

Якщо в класі є захищені дані і захищені функції, то аналогічно С++ такі дані доступні будь-якому насліднику.

**Підтримка поліморфізму в C#**

Поліморфна поведінка забезпечується віртуальними функціями. Для реалізації механізму віртуальної функції використовуються слова virtual I override. Віртуал помічує метод, робить його віртуальним. Оверрайд говорить що такий метод переопрєдєляється в наслєднікє.

В ієрархію класів по працівникам додамо можливість визначення бонуса до зарплати, причлму цей бонус буде визначатися по різнову алгоритму в залежності від типу працівника.

Class employee

//…

Public virtual void gift bonus(float amount)

{

Currpay+=amount;

}

}

Class salesperson:employee

{

Public override void givebonus(floatamount)

{

Int salesbonus;

If(якась умова) salesbonus=10;

Else if(some umova) saesbonus=15;

Else salesbonus=10;

Base.givebonus(amount\*salesbonus);

}

}

Class manager:employee

{

//…

Public override void giftbonus(float amount)

{

Base.givebonus(amount);

Random r=new random();

numberOfOption+=r.next(500);

}

}

В клсі менеджер спочатку викликається гіфт бонус предка, але після цього збільшується поле numberOfOption. Таким чином кожен клас по свєму надає бонус.

Main()

{

Employee emp=new employee();

Manager ch=new manager(“john”,99,…);

Emp=ch;

Emp.giftbonus(300);

Emp.dosplaystatus();

Salesperson sp=new salesperson(“frank”,…);

Emp=sp

Emp.giftbonus(100)

Emp.displaystatus();

}

Запаковування віртуальних членів

Ключове слово сілед моеж застосовуватися не тільки для запаковуання наслідування всього класу, але також для заборони перевизначення конкретної віртуальної функції в наслідниках.

**Абстрактні класи**

Для того щоб деякий базовий клас став абстрактним достатньо написати ключове слово **abstract** пед ключовим словом клас.

Точно так сомо спроба створення екземпляра приведе до помилки компіляції.

Ціль створенння абстрактних класів – явно описати предків майбутньої ієрархії. Наслідування класів чисто віртуальні функції забав'язані бути перевизначені.

**Правила приведення до базового і похідного класу**

1. неявне приведення – означає вскрізь, де очікується посилання на базовий клас можна без всіляких додаткових перетворень вставляти похідний клас. Неявне приведення робиться автоматично.
2. Явне приведення: (тип)посилання;

**Ключове слово** **as**

Явне приведення типу відбувається під час виконання програми. На етапі компіляції воно не перевіряється.

Але під час виконання буде генеруватися виключення.

**Ключове слово is**

Дозволяє визначити сумісність двох типів. На відміну від ес, якщо тип не сумісний – повертається false. Таким чином зазвичай застосування іс в якості оператора іф.

If(emp is Manager){

…

M=emp as manager}

Часто ес та іс використовуються разом. Оточувати оператор приведення типу конструкцією трай кеч непотрібно.

**Батьківський клас всієї ієрархії в .NET**

В мові C# існує головний батьківський клас Object. Навіть коли ми це не вказуємо, він неявно присутній при оголошенні.

Class car{…} = class car:object{…}

В цьому батьківському класі вже визначена частина функціональності, тобто оголошені віртуальні методи, статичні члени і звичайні методи. Тобто оголошуючи деяки клас, частина функціональності вже присутня.

**Інтерфейси в C#**

**Інтерфейс** – набір вичначених дій, які мають бути реалізовані в класі.

Interface – не більш ніж просто іменований набір абстрактних членів. Прийнято імена всіх інтерфейсів починати з букви I.

**Поріняння інтерфейсів і абстрактних базових класів**

Базовий клас стає абстрактним, якщо в ньому визначається хоча б одна чиста віртуальна функція(абстрактний метод). Інтерфейси, в свою чергу, можуть містити визначення тільки абстрактних членів.

Public interface IPointy

{

~~Byte getNumberOfPionts(){return numberOfPoints;}~~

~~Public IPointy(){numberOfPoints=0;}~~

~~Int numberOfPoint;~~

Byte Points{get;}

}

Int main()

{

~~IPointy p=new IPointy();~~

}

Модифікатори доступу до членів інтерфейсів не вказуються. Вони неявно загальнодоступні і абстрактні. Інтерфейси неможуть містити поля даних. Інтерфейси не можуть містити конструкторів. В інтерфейсах неможе міститися реалізація мтеодів. Можна визначати будь-яку кількість властивостей. Інтерфейси являються чистим протоколом., відповідно створювати змінну типу інтерфейсу неможна.

Public interface IPointy{…}

Public class Pencil:IPointy{…}

Public class switchBlate:object, IPointy{…}

Public class Arrow:IClonumber,IPointy{…}

………………………………………………………………..

Class Triangle:Shape, IPointy{

Public Triangle(){…}

Public Triangle(string name; base(name){}

Public override void Drow(){

Console.WriteLine(“Drawing{0} the Triangle”, Petname);}

}

Public byte Points{get{return 3;4}

}

По наслідуванню базхового класу приходить віртуальна функція Draw і ми її перекриваємо. Від інтерфейсу приходить властивість Points і ми зобов’яані йоно перевизначити.

Для перевірки чи підтримує тей чи інший інтерфейс використовуються ключові слова as i is.

**Класи і об’єкти**

Клас з’явився в мові сімула і звстособувався для моделювання реальності. Об’єкт представляє собою конкретний впізнаваний предмет – одиницю або сутність(реальну або абстрактну), що має чітко визначене функціональне призначення в деякій предметній області.

**Об’єкт характеризується**:

1. Станом
2. Володіє певною, добре визначеною поведінкою
3. Володіє унікальною ідентичністю

В ООпідході існує путаниця з ключовим словом object, нарпиклад в мовах pascol, delphі, C# цим словом позначається тип Клас. В ООпідході між класом і об’єктом встановлений наступний зв’язок: об’єкт – екземпляр деякого класу. Об’єкт – змінна. Клас – тип. Клас визначає загальну для деякої групи об’єктів структуру і поведінку.

1. **Стан об’єкту** – характеризується переліком(зазвичай статичних) всіх властивостей даного об’єкту і теперішніми, зазвичай динамічними, значеннями еожної з цих властивостей. Всі влаадреса пам’яті, властивості мають деякі значання. Ці значення можуть бути кількісними хараектеристиками, і можуть посилатися на інший об’єкт – реально зберігається інший об’єкт. В класах за стан об’єкту відповідають дані членів.
2. **Поведінка об’єкту**. Об’єкти не існують окремо, а впливають на інші об’єкти. Поведінка – це те як об’єкт взаємодіє і агрегує. Поведінка виражається в термінах: стан об’єкта і передача повідомлень(виклик якихось функцій). Стан об’єкта представляє собою сумарний результат його поведінки. Поведінка – спостерігаєма і перевіряєма із зовні діяльність об’єкту. Передача повідомлень може називатися виконанням операції на об’єктом, виклик функції і виконання методів.

class Queue

{

Public:

Queue();

Queue(Queue&);

~Queue();

Queue& operator=(const Queue&);

Bool operator==(const Queue)const;

Bool operator!=(const Queue)const;

Virtual void clear();

Virtual void append(const Void\*);

Virtual void pop();

Virtual void remove(int at);

Int length()const;

Bool isEmpty()const;

Virtual const void\* front()const;

Protected:

…

}

**Всього 5 видів операцій:**

1. Модифікатор – змінює стан об’єкту. В даному прикладі clear pop куьщму.
2. Селектор – зчитує значення, але не змінює його(length, isEpmty, Length);
3. Ітератор – операція, що дозволяє реалізувати доступ до всих частин об’єкту в строго визначеній послідовності.
4. Конструктор
5. Деструктор

В деяких мовах програмування конструктор – частина опису класу. В інших(SmallTalk) – описуються в спеціальних мета класах.

В даному прикладі вказано, показана поведінка, що моделюється за допомогою членів класів. Поведінка може бути реалізована за допомогою функцій, що не входять в жоден клас. Вони називаються глобальними або вільними процедурами. Але в деяких мовах програмування вільної процедури описати неможна. Існують статичні функції, що існують навіть без екземплярів даного класу. Такі функції можна викликати, навіть не маючи об’єкту.

**Ідентичність**

Ідентичність об’єкту – така властивітсь об’єкту, що вдірізняє конкретний об’єкт від всіх інших об’єктів.

Struct Point

{

Int x; int y;

Point():x(0),y(0){};

Point(int xval, int yval):x(xval),y(yval){

}

};

Class displayItem{

Public:

displayItem();

displayItem(Point& location);

displayItem();

virtual void draw();

virtual voidselect();

virtual voidinSelect();

virtual void move(const point& location);

bool isSelected()const;

Point location()const;

Bool isUnder(const Point& location)const;

}

**Відношення між класами**

**Наслідування**

При наслідуванні може бути конфлікт імен. В класах предках можуть бути поля з однаковими іменами. Виникає питання єкий тип, скільки штук і т.д.

Це може бути вирішено за допомгою:

1. Помилка при компіляції(smalltolk, eifel) в ейфелі можна штучно перейменувати.
2. Вважати що однакові імена означають однаковий атрибут(CLOS).
3. В насліднику будуть присутні обоє імен. Додається уточнення у вигляді додавання імені базового класу a::value, b::value.

Може виникти проблема при повторному наслідуванні. Тобто наслідування по колу.(ромбовидне або повторне наслідування). Виникає питання скільки разів дані члени, оголошені в класі А присутні в класі Д.

Можна вирішити ртьома способами:

1. Помилка при компіляції
2. В класі Д будуть присутні дві копії наслідуваного елементу. Вони будуть розрізнятися шляхом вказання префіксів класу(С++).
3. В класі Д тільки одна копія даних і функцій, що оголошені в А. але для цього, при оголошенні, один із класів має бути віртуальним(С++).

Class D: public B, virtual public C

{

…

}

**Агрегація**

Відношення агрегації між класами тісно пов’язано відношення агрегаці між об’єктами.

Class TemperatureControl

{

Public:

TemperatureController(Location);

~TemperatureController();

Void process(const Temperature Ramp&);

//…

Private: Heater h;

}

**Використання**

Відношення використання показує зв’язок типу клієнт-сервер.

Клас TemperatureRamp згаданий як частина оголошення функції члену schedule. Це означає що клас темперчеконтроллер користується послугами темперчеремп. Механізм інкасуляції дозволяє отримати доступ тількиа для відфкритої частини серверу. Іноді для оптипіхаціцї і використання роботи, функції роблять дружніми.

**Інстанціювання**

Це таке відношення між класами, що виявляється між шаблонами.

Template <class Item>

Class Queue{

Public:

Queue();

Queue(const Queue <Item>&);

~Queue();

Virtual void clear();

Virtual void append(const Item&);

Virtual void remowe(int at);

Virtual int length();

Virtual bool isEmpty();

Virtual bool push(item&);

Vittual item pop();

Protected:

Item\* array;

}

Queue <int> intQueue;

Queue <DisplayItem> itemQueue;

**Шаблони класів**

Онструктор поза класом

Template <class T>

Stack <T>::stack(ints)

{

Size = (5>0 && s<1000)?5:50;

Top=-1; stackPtr=new T[size];

}

Стек – масив, що займає неперервну частину пам’яті.

Деструктор не повинен очищати розмір стеку.

**Використання шаблону**

Main()

{

Stack<float> floatStack(5);

Float f==1.1;

While(floatStack.push(f))

{

Cout<<f;

}

}

Відмінністю шаблонів класів від шаблонів функцій в тому що при оголошення змінної явно вказується тип, замість мітки заповнювача шаблону.

**Три типи дружності шаблонів**

1. Emplate<class T> class X{…
   1. friend void f1(); - дружня для всіх.
   2. Friend void f2(T&); тільки для тих типів, які співпадають з типом класу.
2. Nnn
   1. friend void A::f3(); для будь-якого класу, отриманого з X.
   2. Friend void c<T>::f5(x<T>&); член класу С буде дружня лише одному шаблонному класу, отрманому з Х того ж параметру Т.
3. Ттт
   1. friend class Y; Y буде дружнім будь-якому клсу, отрманому з Х
   2. friend class Z<T> дружній лише тому класу, де співпадають параметри

**статичні дані для ріхних параметрів шаблонів** не об’єднуються. Тобто кожен шаблонний клас, отиманий із деякого шаблону Х отримує власну копію статичних даних і функцій.

**Особливості шаблонів у C#**

В літературі по С№ термін шаблону не використовують. Використовують узагальнені методи і класи.

Узагальнені методи

Static void Swap<T>(ref T a, ref T b)

{

Console.writeline(“{0}”typeof(T);

T temp; temp=a; a=8; b=temp;

}

Функція змінює передані параметри по значенню, тому потрібно написати ref це теж, що і &. Вказання Т в кутових скобках каже те, що метод узагальнений.

Main(){

int aa=10,bb=90;

swap <int>(ref aa, ref bb);

string s1=”hello”, s2=”true”;

swap <string>(ref s1, ref s2);

екран

90, 10

There hello

Особливістю узагальненого методу є те, що при виклику інформації по конкретному типу може бути явно вказана. Рукомендується не упускати <int>