Об’єктно-орієнтоване програмування

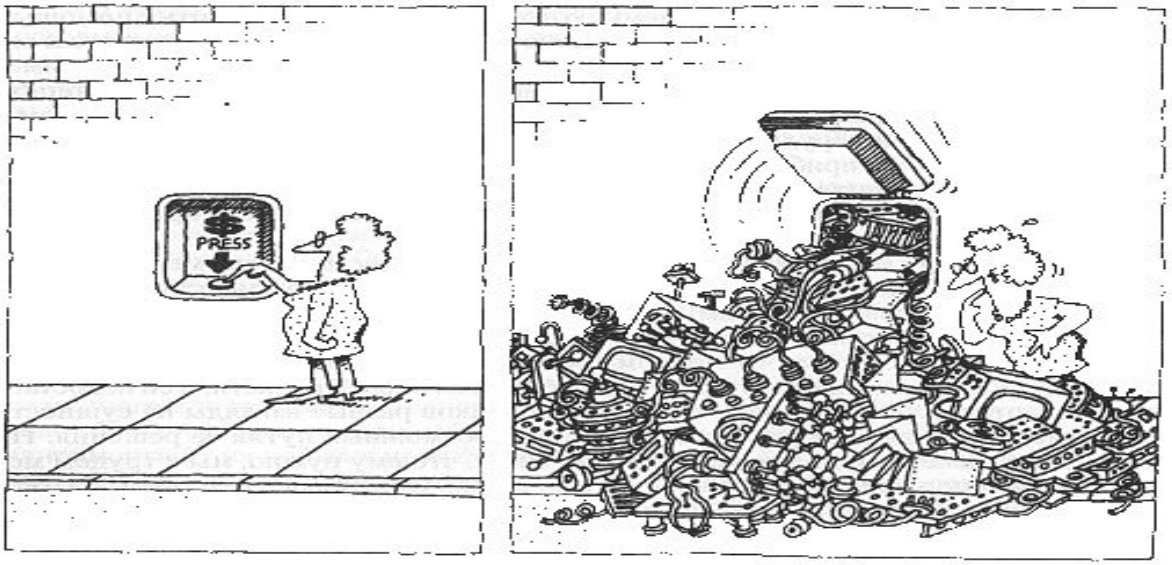
Лекція №1

1. Предмет та задачі курсу

- ***Предмет:*** вивчення концепцій об'єктно-орієнтованого проектування (OOD) та програмування (OOP).
- ***Область застосування OOD та OOP*** — складні програмні продукти.
- ***Прості програми*** — це програми, які розробляються та використовуються однією людиною . Властивості:
  - мають обмежений час існування;
  - мають обмежену область застосування;
  - їх простіше переписати, ніж розширити.
- ***Промислові програмні продукти — складні***, використовуються для вирішення різних задач (системне керування зворотним зв'язком, обробка БД з паралельним доступом, поновленнями та запитами, контроль за реальними процесами — диспетчери). Їм властиві:
  - значний час існування;
  - багато користувачів;
  - високий рівень складності — для розробки програмного забезпеченя потрібно декілька осіб, так як його складність перевищує можливості інтелекту однієї людини.

2. Чинники складності програмного забезпечення (ПЗ)

- складність реальної предметної області:
  - несумісність між замовниками та розробниками, оскільки вони спеціалісти у різних галузях. Різні погляди на сутність проблеми – зміни вимог у процесі розробки, часто після отримання перших результатів;
- складність керування процесом розробки:
  - необхідно створити ілюзію простоти в процесі роботи команди;
- необхідність достатньо гнучкої системи:
  - створення спеціалізованих бібліотек;
- складність опису поведінки великих дискретних систем:
  - в великій програмі існують тисячі змінних та потоків керування.



Задача розробника – створити ілюзію простоти

3. Шлях розв'язання складних задач - об'єктний підхід (1/2)

- **Розв'язання складних задач грунтується на наступних принципах ООП:**
  - Декомпозиція – поділ задачі на складові
  - Абстракція – виділення головного
  - Ієрархія
- **1. Декомпозиція** – дозволяє тримати в пам'яті інформацію лише про деякі частини.
  - ***Алгоритмічна декомпозиція*** — це структурне програмування "згори-донизу".
    - **Алгоритмічна декомпозиція** Концентрує увагу на послідовності дій — підпрограми. Структурний підхід не працює для великих програм (умовно, коли програма містить більше, ніж 100000 рядків).
  - ***Об'єктно-орієнтована декомпозиція*** — це програмування за належністю елементів до різних абстракцій; поділ не на *кроки*, а на *об'єкти*. Кожний об'єкт має власну поведінку.
    - **Об'єктно-орієнтована декомпозиція** Концентрує увагу на об' єктах чи суб'єктах дії. Зменшує розмір програми за рахунок повторного використання різних механізмів. Більш гнучкі та легко еволюціонують, розвиваються з менших систем.
    - Важливі обидва аспекти: як алгоритмічна, так і об'єктно-орієнтована декомпозиція.

3. Шлях розв'язання складних задач - об'єктний підхід (2/2)

- **Приклади**:
  - а. Нейронні мережі:
    - *Алгоритмічна декомпозиція*: навчання, робота, забування.
    - *Об'єктно-орієнтована декомпозиція*: шари нервових клітин містять ансамблі нейронів.
    - *Функція об'єкту*: навчання.
  - б. Транснаціональна корпорація:
    - *Алгоритмічна декомпозиція*: виробництво, продаж, участь у політичній діяльності (вибори).
    - *Об'єктно-орієнтована декомпозиція*: транснаціональні корпорації складаються з національних компаній, які містять відділення та філіали (відділи збуту).
    - *Функція* об'єкту: політичне життя.
- 2. ***Абстракція*** — людина може одночасно сприйняти 7±2 одиниці інформації. Тому необхідно визначати **головну частину інформації (абстракція).** В такому разі можна використовувати інформацію більшого семантичного об'єму.
- 3. ***Ієрархія*** — дозволяє вивчати механізми взаємодії компонентів. Поведінка кожного окремого об'єкту відповідає поведінці свого рівня (наприклад клітини). Легко визначити загальні та особливі риси.
  - В програмній системі дуже *важко* визначити ієрархію. Але після цього структура системи прояснюється.

Література

- 1. Гради Буч “Объектно-ориентированное проектирование с примерами приложений на C++”
- 2. Айра Пол «Объектно- ориентированное программирование на C++»
- 3. Герберт Шилдт «Самоучитель по C++»
- 4. Стивен Прата «Язык программирования C++»
- 5. А также, как и и ранее **Бьерн Страуструп**

Об’єктно-орієнтоване програмування

Лекція №2

Еволюція об'єктної моделі

Еволюція об'єктної моделі зумовлена швидким розвитком програмування:

- Зміщення акцентів від програмування окремих деталей до програмування більш крупних компонент;
- Розвиток мов програмування вищого рівня.

Розвиток ООП

Розвиток ООП є *еволюційним* розвитком проектування.

- Передумови розвитку об’єктного підходу:
- прогрес в області архітектури ЕОМ (поява об’єктно-орієнтованої архітектури ЕОМ);
- розвиток мов програмування;
- розвиток методології (стилю) програмування: модульність та приховання даних;
- розвиток теорії баз даних (ENTITY-RELATIONSHIP-моделі, сутність-відношення);
- штучний інтелект (теорія фреймів, ансамблів);
- філософія та теорія пізнання — світ можна розглядати в термінах об’єктів та подій. Розум людини — сукупність різноманітно мислячих агентів.

Реалізація ООП

Процес реалізації ОО підходу (макропроцес):

- Концептуалізація (встановлення основних вимог);
- Аналіз (побудова моделі відповідної поведінки);
- Проектування (створення архітектури);
- Супроводження (керування еволюцією після виходу).

Мікропроцес:

- Ідентифікація класів та об’єктів на даному рівні абстракції;
- Ідентифікація семантики класів та об’єктів;
- Ідентифікація відношень між класами;
- Специфікація інтерфейсів та початок реалізації.

Об’єктно-орієнтоване програмування

ОО Програмування — це методологія програмування, що базується на представленні програми у вигляді сукупності 1) *об’єктів*, кожний з яких є екземпляром 2) *класів*, а класи створюють 3) *ієрархії*.

- Програма є об’єктно-орієнтованою при наявності всіх 3-х вимог.

Характеристики ОО мови

Мова є об’єктно-орієнтованою, якщо:

- Підтримує об’єкти (інтерфейс у вигляді іменованих операцій та власні дані з обмеженим доступом);
- Об’єкти відносяться до відповідних типів (класів);
- Типи (класи) можуть успадковувати атрибути супертипів (відношення спадкування “is a” “це є “).

ООП

- *Об’єктно-орієнтоване проектування* (OOD — Object Oriented Design) — це методологія проектування, що поєднує в собі процес об’єктної 1) *декомпозиції* та способи представлення 2) *логічної, фізичної, статичної* та *динамічної* моделей системи.

1. Декомпозиція — об’єктна;
2. Багато засобів представлення моделей.

ОО аналіз

Об’єктно-орієнтований аналіз (ООА — Object Oriented Analysis) — це методологія, при якій вимоги до системи сприймаються з точки зору класів та об’єктів, виявлених в предметній області. Під цим терміном розуміють об’єктно-орієнтований світогляд.

Складові частини ОО підходу

- Абстракція
- Інкапсуляція
- Модульність
- Ієрархія

Приклад

```
#include <iostream.h>
//Make public and private explicit in class
class complex_c1 {
 public://need to know style- our preference
    void assign(double r, double i);
    void  print()
       { cout << real << " + " << imag << "i ";}
 private:
    double  real, imag;
 };
inline void complex_c1::assign(double r, double i = 0.0)
 {
    real = r;
    imag = i;
 }
```

Абстракція

- Абстракція — це відмова від зайвих деталей класу об`єктів та виділення важливих деталей їх структури та поведінки.
- Інтерфейс об’єкту містить важливі аспекти поведінки. Абстракція дозволяє реалізувати відношення між об’єктами: *клієнт-сервер* — один об’єкт (клієнт) використовує ресурси іншого об’єкта (сервера), на основі якої формується контрактна модель програми, тобто внутрішня структура об’єкта визначається контрактом.

Зовнішня взаємодія — контракт:

- 1. Кожна операція задається формальними параметрами та типом значень, що вертається;
- 2. Повний набір операцій — протокол (містить всі засоби) — *поліморфізм*;
- 3. Інваріант — деяка логічна умова, значення якої повинно зберігатися (передумова та постумова).

Приклад

```
#include <iostream.h>
//Make public and private explicit in class
class complex_c1 {
 public://need to know style- our preference
    void assign(double r, double i);
    void  print()
       { cout << real << " + " << imag << "i ";}
 private:
    double  real, imag;
};
inline void complex_c1::assign(double r, double i = 0.0)
{
    real = r;
    imag = i;
}
```

Інкапсуляція

Інкапсуляція – це процес відокремлення одного елементу від інших елементів об’єкту, що визначають його структуру від поведінки. Призначена для того, щоб ізолювати контрактні обов’язки абстракції від їх реалізації $\Rightarrow$ дозволяє створити уявність простоти програми.

Приклад

```
#include <iostream.h>
//Make public and private explicit in class
class complex_c1 {
 public://need to know style- our preference
    void assign(double r, double i);
    void  print()
       { cout << real << " + " << imag << "i ";}
 private:
    double  real, imag;
};
inline void complex_c1::assign(double r, double i = 0.0)
{
    real = r;
    imag = i;
}
```

Модульність

Модульність — це властивість системи, що була розкладена внутрішньо, в якій модулі пов’язані між собою слабко.

Модулі — об’єктні:

- виконують роль фізичних контейнерів, що містять визначення класів та об’єктів;
- розроблюються різними людьми;
- модулі можуть компілюватися окремо;
- в один модуль групуються логічно пов’язані класи та об’єкти.

Приклад

```
#include <iostream.h>
//Make public and private explicit in class
class complex_c1 {
 public://need to know style- our preference
    void assign(double r, double i);
    void  print()
       { cout << real << " + " << imag << "i ";}
 private:
    double  real, imag;
};
inline void complex_c1::assign(double r, double i = 0.0)
 {
    real = r;
    imag = i;
 }
```

Ієрархія

Ієрархія — це впорядкування абстракцій та розміщення їх по рівням.

Основні види ієрархії:

- 1. Структура класів “is a” (узагальнення-спеціалізація)
- 2. Структура об’єктів “part of” (агрегація структури [включення])

Приклади ієрархії

- Одиночне спадкування — від одного класу;
- Множинне спадкування — від багатьох (декількох класів).
- Спадкування — це ієрархія “узагальнення-спеціалізація”.

ОСНОВНІ ВЛАСТИВОСТІ ООП

До основних принципів ООП відносяться наступні:

1. *Абстракція* — виділення головного.
2. *Інкапсуляція* — можливість приховати внутрішні деталі під час опису загального інтерфейсу. *Ключові слова* private, public, protected визначають рівень доступу для забезпечення інкапсуляції (на рівні об'єктів);
3. *Успадкування* — передача властивостей від попередника (отримання властивостей з протоколу базового класу). Засіб отримання нових класів з існуючих;
4. *Поліморфізм* — умови, в яких вид має різні морфологічні форми. Окремі повідомлення можуть викликати різноманітні дії. Спосіб реалізації — перевантаження операцій та функцій – паралельний поліморфізм, коли тип невизначений, а його значення вказується пізніше (в C++ — базові вказівники та шаблони).

Основна концепція ООП

Основна концепція ООП — передача повідомлень об'єктам.

Абстрактні типи даних в C++ реалізуються за допомогою механізму класів (дозволяє керувати видимістю того, що є в основі виконання).

- *Клас* — ключове слово "Class" — абстрактний тип даних.
- *Об'єкт* — змінна типу ClassName, де ClassName — визначений раніше клас.
- *Дані стану* оголошуються в описі класу і називаються полями даних, дані-члени чи члени.

Приклад

```
main()
{
   complex_c1  b;
   complex_c1 * c;
   b.assign(3.3, 4.4);
   c->assign(5.5, 6.6);
   cout << "\nComplex from class definition: ";
   b.print();
   cout << "\nComplex from explicit class definition: ";
   c.print();
}
```

Основна концепція ООП

- *Повідомлення* – вказуються за допомогою прототипів функцій в описі класу. *Прототип функції* — це тип значення, що повертається, ім’я функції та список параметрів. *Список параметрів* включає типи параметрів та необов’язкові імена параметрів.
- *Метод* в C++ — це визначення (реалізація) функції. Прототипи функції та визначення називаються функціями-членами класу. *Члени класу* — це поля даних та функції-члени.
- *Підклас* — похідний клас. Його суперклас чи базовий клас називається базовим.

Лекція 3.

Класи та об’єкти. Проектування класів та їх методів.

Мета введення класів в C++

- Головна мета введення концепції класів в C++ — це забезпечення програміста засобами для створення нових типів, які були б такими ж зручними у використанні, як і вбудовані.
- **Тип** — конкретний представник деякої концепції.
- **Клас** — це тип, що визначає користувач. Для визначення концепції,

Функції-члени

- Функції, визначені всередині опису класу (до речі, структура — це один з видів класу), називаються ***функціями-членами***, і їх можна викликати тільки для змінної відповідного типу, використовуючи стандартний синтаксис доступу до членів структури.
- Оскільки різні структури можуть мати функції-члени з однаковими назвами,

Приклад

```
class complex_c1 {
 public://need to know style- our
   preference
    void assign(double r, double i);
    void  print()
       { cout << real << " + " << imag <<
   "i ";}
 private:
    double  real, imag;
};
```

Контроль доступу

Розглянемо приклад:

```
class X
 {
public:
    void init();
    int getITnow();
private:
    int x,y,z;
```

Контроль доступу

- Імена в закритій private частині можна використовувати тільки у функціях-членах класу. Відкрита public частина утворює відкритий інтерфейс об’єктів класу. (Структура — клас, члени якого відкриті за замовчуванням). Крім того, існує мітка protected (захищений), тобто всі змінні будуть доступні тільки прямим нащадкам цього класу.
- Захист закритих даних базується на

Статичні члени

Змінну, яка є частиною класу, а не частиною об’єкта цього класу, називають ***статичним членом*** і позначають специфікатором **static**

Приклад

```
#include <iostream>
    using namespace std;
    class Conscription {
      static int Age;
    public:
      void setInt(int n) {
         Age = n;
      }
      int getInt() {
         return Age;
      }
    };

    int main()
    {
       Conscription  Petrov, Sidorov;

      Petrov.setInt(18);

      cout << “Petrov’s age: " << Petrov.getInt() << '\n'; // displays 18
      cout << “Sidorov’s
    age: " << Sidorov.getInt()<<'\n';// also displays 18

      return 0;
    }
```

Константні функції-члени

Нехай у класі X існують функції, які надають і змінюють значення об’єкта типу X. Але, на жаль, не існує способу для перевірки значення об’єкта X. Проте цю проблему можна легко вирішити, описавши ці функції як ***константні функції-члени***, тобто функції, які не змінюють стан X:

Приклад: три способи використання const

```
class Birthday {
int d,m,y;
const string congratulations;
            public:
              int day() const {return d;}
              int month() const {return m;}
              int year() const {return y;}
          const string getCon() { return congratulations; }
         void print (const int d, const int m, const int y)
          {cout<<“My birthday is ”<< d<<“.”<<m<<“.”<<y<<“.”;}
            //…
            };
```

Константні функції-члени

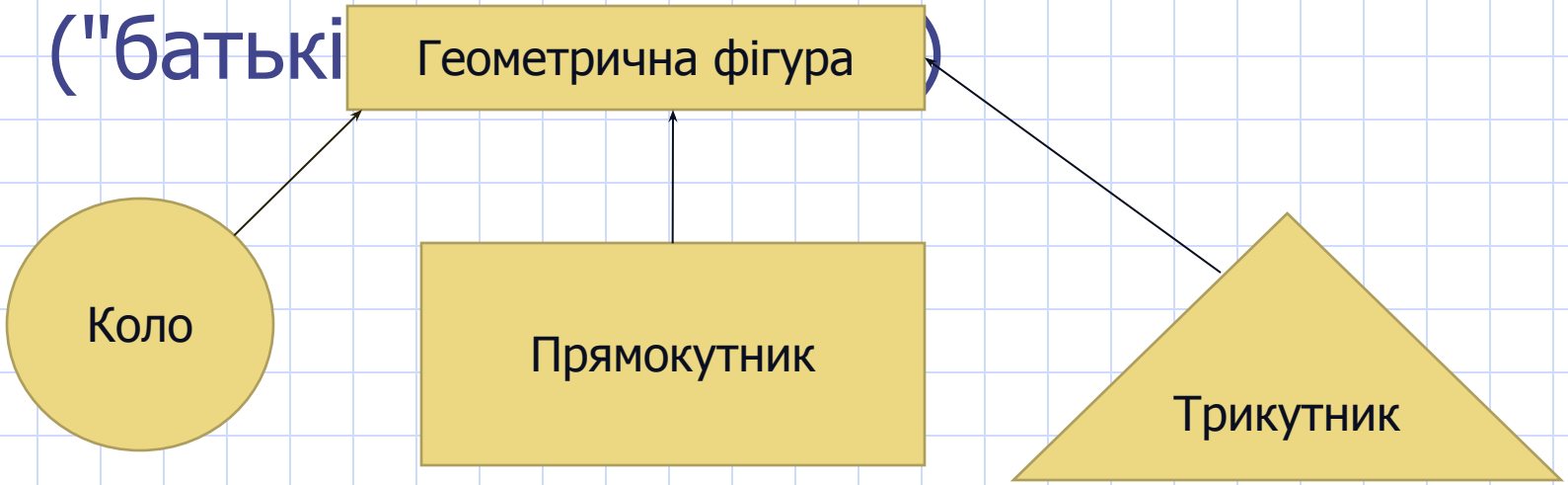
- Коли константна функція-член описується зовні, а не всередині класу, то потрібно додати суфікс const:

```
inline int Date::year() const
//правильно
{ return y;}
```

- Константну функцію-член можна

Підкласи

Підкласи — це класи, які успадковують усі "властивості" суперкласу ("батькі



Віртуальні функції

Віртуальні функції визначаються специфікатором **virtual** і дозволяють програмісту описати в базовому класі функції, які можна було б замінити у кожному наступному класі.

Ієрархія класів

- Об’єкти різних класів і самі класи можуть перебувати у відношенні успадкування, за якого формується ієрархія об’єктів, що відповідає заздалегідь передбаченій ієрархії класів.
- Ієрархія класів дозволяє визначати нові класи на основі вже існуючих. Існуючі класи зазвичай називають ***базовими*** (інколи ***батьківським***), а нові класи, що формуються на основі базових, — ***похідними*** (***породженими***), інколи ***класами-нащадками*** або ***спадкоємцями***. Похідні класи “отримують спадок” — дані і методи своїх базових класів — і, крім того, можуть поповнюватись власними компонентами (даними і власними методами).
- Наприклад, за таким визначенням

  class S: X{…};

  клас S породжений класом X, звідки він успадковує компоненти.

Приклад виконання

```
#include <string>
#include <conio.h>
#include <iostream>

using namespace std;


class Student
{
 private:
   string name;
   int age, course;
 public:
   void setData();
   void getData();
  };
```

Приклад виконання

```
void Student::setData()
{
 cout<<"Enter name"<< endl;
 cin>>name;
}
void Student::getData()
{
 cout<< " Name=" <<name<<";"<<endl;
 cout<< " Age=" <<age;
}
```

Приклад виконання

```
//-------------------------------------------
int main()
{
  Student Olga;   //створення об’єкта
  Student* Nick;  //створення вказівника
  Olga.setData(); //виклик функції setData() об’єкта Olga
  Olga.getData();
  Nick=new Student; //виділення пам’яті під об’єкт
  Nick->setData();  //виклик функції об’єкта *Nick
  Nick->getData();
  getch();
        return 0;
}
```

Об'єкти

- *Об'єкт завжди:*
  - є чітко обмеженим
  - не обов'язково є відчутним
- **Приклад**: процес хімічного виробництва
- *Об'єкт* характеризується станом, поведінкою та ідентичністю;

структура та поведінка схожих об'єктів визначають їх загальний *клас*. Об`єкт - це екземпляр класу.

Приклад

```
class PersonnelRecord
   {
      public:
         char * employee Name () const;
   //всі об'єкти можуть
         int employee Local Security Number
   () const;
                                        //отримати
   дані
         char * employee Department ()
   const;
```

Поведінка об’єкту

- *Поведінка об‘єкту* — це спосіб дії та реакції об‘єкту.
- Поведінка висловлюється в термінах стану об‘єкту та передачі повідомлень.
- *Поведінка* — яка спостерігається і перевіряється зовні дії об‘єкту.
- **Приклад**: Автомат — в залежності від стану (кількості монет) видає або ні напій.

Приклад

- **Розглянемо клас «Черга»**
- ```
  class Queue
  {
    public:
      Queue ();
      Queue (const Queue &);
      virtual ~Queue (); //знищує чергу, але не її учасників
      virtual Queue & Operator = (const Queue &)
                          //virtual - буде визначатися
                          // в класах-нащадках
      virtual int operator = = (const Queue &) const;
      int operator! = (const Queue &) const;
      virtual void clear ();
      virtual void append (const void *);
      virtual void pop ();          //просування черги
      virtual void remove (int at);
      virtual int length ();
      virtual int isEmpty () const;
      virtual const void * front () const;
      virtual int location (const void *)const;
    protected:

  };
  ```
  ```

Mutable

mutable — антипод const, визначає член, який не буде const ні за яких умов (навіть для const об'єкта).

Відношення між класами

Класи не існують ізольовано.

Основні типи відносин між класами:

1. "узагальнення/спеціалізація" (загальне–частинне) “is a”
2. "ціле–частина" — “part of”
3. асоціація — семантичний, смисловий зв`язок.

ОО мови програмування підтримують різні комбінації наступних типів відносин:

- асоціація — найбільш загальне та невизначене відношення
- успадкування — "загальне–частинне"
- агрегація — "ціле–частина"
- використання — наявність зв`язку між екземплярами класів
- інстанціонування — специфічний різновид узагальнення
- метаклас — клас класів (класи як об`єкти).

Приклад

Приклад

Студент КПІ - студент

Студент мед. Університету - студент

Студенти фіз-теху та ФІОТу - студенти КПІ

Відмінники - складові частини обох типів студентів

Викладачі - наводять жах на

Асоціація

- **Приклад** — товари та продаж.
- Class Product; //те, що продали
  Class Sale; //угода, в якій продано
  //декілька товарів

```
Class Product {
  public:
    . . .
  protected:
    Sale* last Sale;
};
Class Sale {
  public:
    . . .
  protected:
    Product** product Sold;
};
```

Асоціація

- *Асоціація* — смисловий зв'язок, як правило, не має напрямку та не пояснює, як класи спілкуються один з одним.
- *Потужність* — кількість учасників цього смислового зв`язку
  - один до одного;
  - один до багатьох;
  - багато до багатьох.
- *Агрегація* — включення одного класу до іншого — відповідає *агрегації* між екземплярами.
- Агрегація як співвідношення "*ціле–частина*" є *спрямованою*. Не вимагає *обов'язкового* фізичного включення (акціонер *володіє* акціями, але не складається з них).
- Якщо (і тільки якщо) існує відношення "ціле–частина" між об'єктами, класи повинні знаходитися у співвідношенні *агрегації*.
- *Використання* — спрямовано, якщо клас є частиною сигнатури функції-члена іншого класу (параметром).
- Використання класів –> рівноправний зв'язок між їх екземплярами (*клієнт–сервер*).

Конструктори

Одне з основних завдань об’єктно-орієнтованого програмування полягає у тому, щоб об’єкти описаного раз і назавжди класу працювали «правильно» — тобто так, як це визначає модель. Кожний об’єкт перед тим як почати роботу, потрібно створити, тобто перевести в якийсь початковий стан. Отже, треба якимось чином описати можливі механізми створення об’єктів даного класу. Для

Приклад. Конструктор з параметрами для класу «Point»

```
class Point
{
  public:
    Point(int x0, int y0);
  private:
    int x, y;
};
Point::Point(int x0, int y0)
{
  x=x0;
  y=y0;
}
```

Тепер для створення об’єкта класу Point потрібно після імені змінної вказати параметри, як для виклику функції:

```
Point A(1, 1), B(2, 0);
```

Типи конструкторів

- Існують деякі типи конструкторів, які, крім безпосереднього використання, автоматично викликаються у деяких особливих ситуаціях.
- **Конструктор за замовчуванням**
- Конструктор за замовчуванням - це конструктор, що викликається без параметрів:

  Point();

  Point(int a=5);
- Його використовують для створення масиву об’єктів, оскільки не зрозуміло, які конструктори і з якими параметрами треба викликати для кожного елементу масиву. Наприклад:

  Point A[10];

  Point* B=new Point[10];
- Конструктор за замовчуванням викликається також тоді, якщо не вказано параметри для ініціалізації об’єкта, як у цьому випадку:

  Point p;

Конструктор копіювання

- Цей конструктор викликається тоді, коли потрібно створити копію об’єкта. Аргументом цього конструктора має бути посилання на об’єкт цього самого класу:

```
Point(Point& p);
```

- Важливим випадком, коли викликається конструктор копіювання, є передавання об’єкта у функцію як параметра за значенням. Тоді

Приклад. Клас String з реалізованими конструкторами

```
class String
{
public:
  String();                       // конструктор за замовчуванням
  String(const String& s); // конструктор копіювання
  String(const char* s);      // конструктор з параметром
                                          // const char*, який являє собою
                                          // стандартний рядок  s
  ~String();              // деструктор
private:
  char* array;                 // масив символів
  int size;                       // розмір масиву
};
```

Приклад виклику конструкторів

```
int main()
{ String a, b;  // конструктор за
    замовчуванням
  String c(a);  // конструктор копіювання
  print(a);     // конструктор копіювання,
   оскільки
                // аргумент передається у
   функцію за значенням
  String d("One");   // конструктор з
```

Деструктори

Конструктори ініціалізують об’єкт, тобто вони створюють середовище, у якому "працюють" функції-члени. Іноді створення такого середовища зумовлює "захоплення" якихось ресурсів: пам’яті, файлу, процесорного часу, які повинні бути "звільнені" після їх використання. Тобто класам потрібна функція, яка б знищувала об’єкт аналогічно тому, як його створює конструктор. Такі функції називають

Приклад деструктора

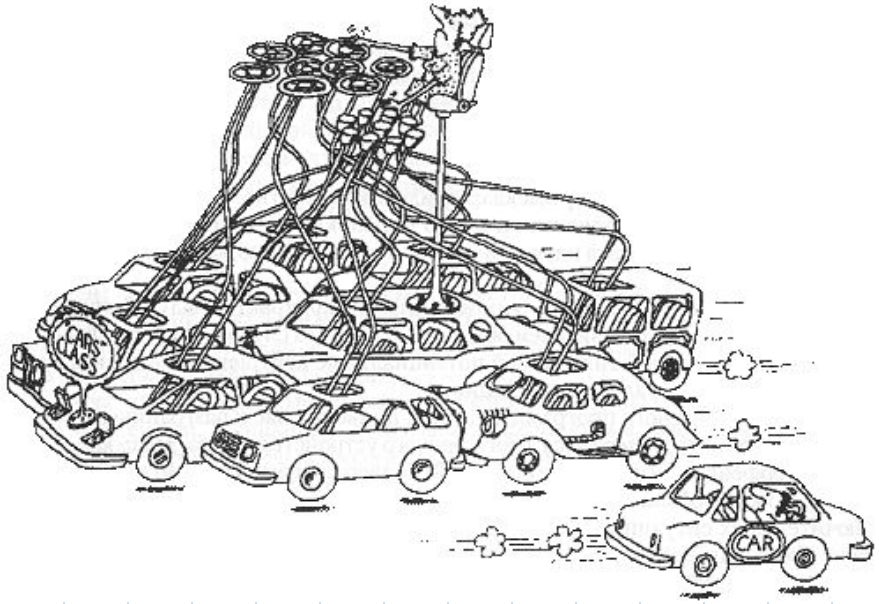
```
class Column
{
    const char* s;
    // ...
};

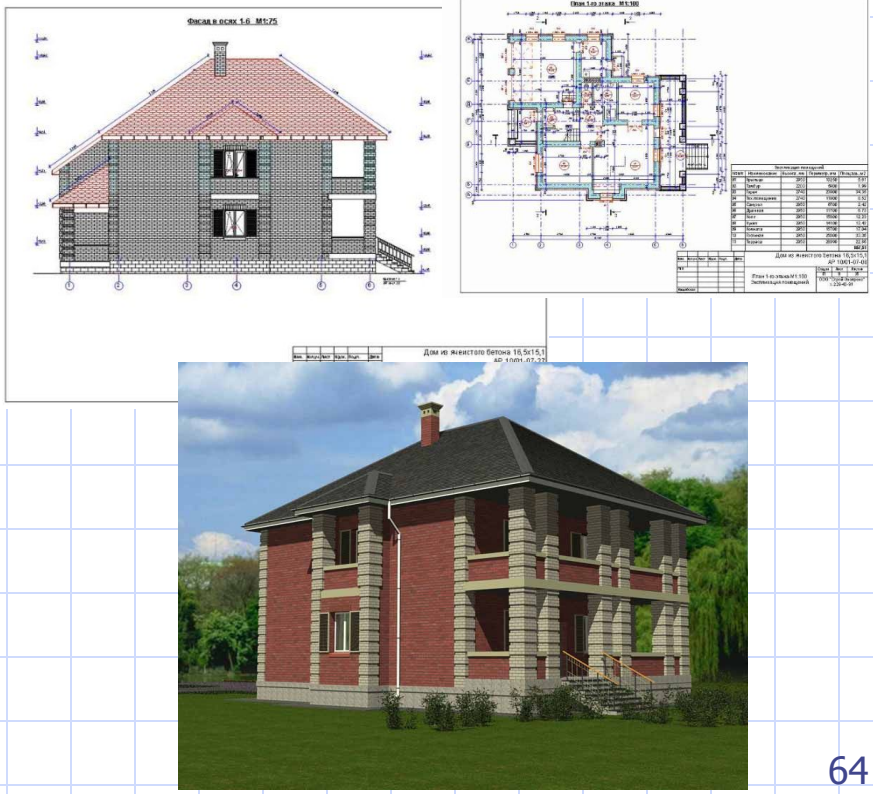
class Table
{
   Column* p;
   size_t sz;
 public:
   Table(size_t s=15) {p=new Column[sz=s];}
//конструктор
  ~Table() {delete[] p;}                    //деструктор
// ...
};
```

Лекция 4. Отношения между классами

Классы и объекты

- **Класс - это некое множество объектов, имеющих общую структуру и общее поведение**





Классы и объекты

- **Объект обозначает конкретную сущность, определенную во времени и в пространстве**

Млекопитающие

...

...

Типи відношень

Класи, як і об`єкти не існують ізольовано!!!

Основні типи відношень між класами (на етапі проектування)

- “Узагальнення/спеціалізація" (загальне–частинне) “is a”
  - напр., троянди є частинним випадком квітів, тобто підкласом більш загального класу «квіти»
- Ціле/частина("part of")
  - напр., лепестки являются частью цветов
- Семантичний, змістовний зв’язок та асоціація
  - напр., насекомые ассоциируются с цветами

Отношения в языках программирования 1/2

ОО языки программирования поддерживают разные комбинации следующих отношений

- **Ассоциация**
  - обозначает смысловую связь между классами
  - наиболее общее и неопределенное отношение
- **Наследование**
  - выражает отношение общего и частного
  - напр., цветок – роза, человек – студент
- **Агрегация**
  - выражает отношение целого и части
  - напр., цветок состоит из лепестков,

Отношения в языках программирования 2/2

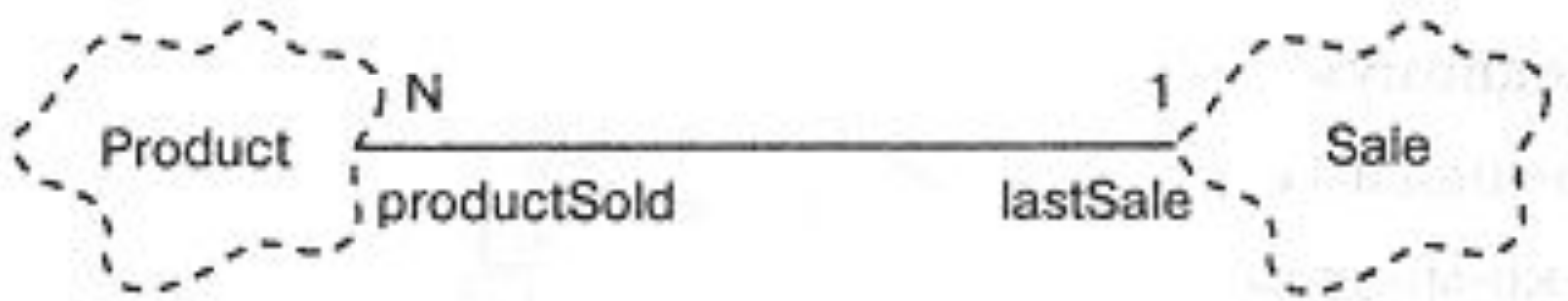
- **Использование**
  - наличие связи между экземплярами классов
  - напр., экземпляр класса явл. параметром функции-члена другого класса
- **Инстанцирование**
  - специфическая разновидность обобщения
  - напр., в C++ - шаблоны
- **Метакласс**
  - это класс классов, что позволяет трактовать классы как объекты

Ассоциация

- Отношение, означающее **некоторую смысловую** связь между классами
- *Асоціація* — смисловий зв'язок, як правило, не має напрямку та **не пояснює**, як класи спілкуються один з одним.
- *Потужність* — кількість учасників цього смислового зв`язку
  - "один до одного";
  - "один до багатьох";
  - "багато до багатьох".

Ассоциация

- **Пример 1** – система розничной торговли
  - класс **Product** - то, что продано в некоторой сделке
  - класс **Sale** - сама сделка, в которой продано несколько товаров



Асоціація

Приклад — товари та продажі.

```
class Product;   //те, що продали
class Sale;      //угода, в якій продано
                 //декілька товарів

class Product {
  public:
    . . .
  protected:
    Sale* last Sale;
};

class Sale {
```

Ассоциация

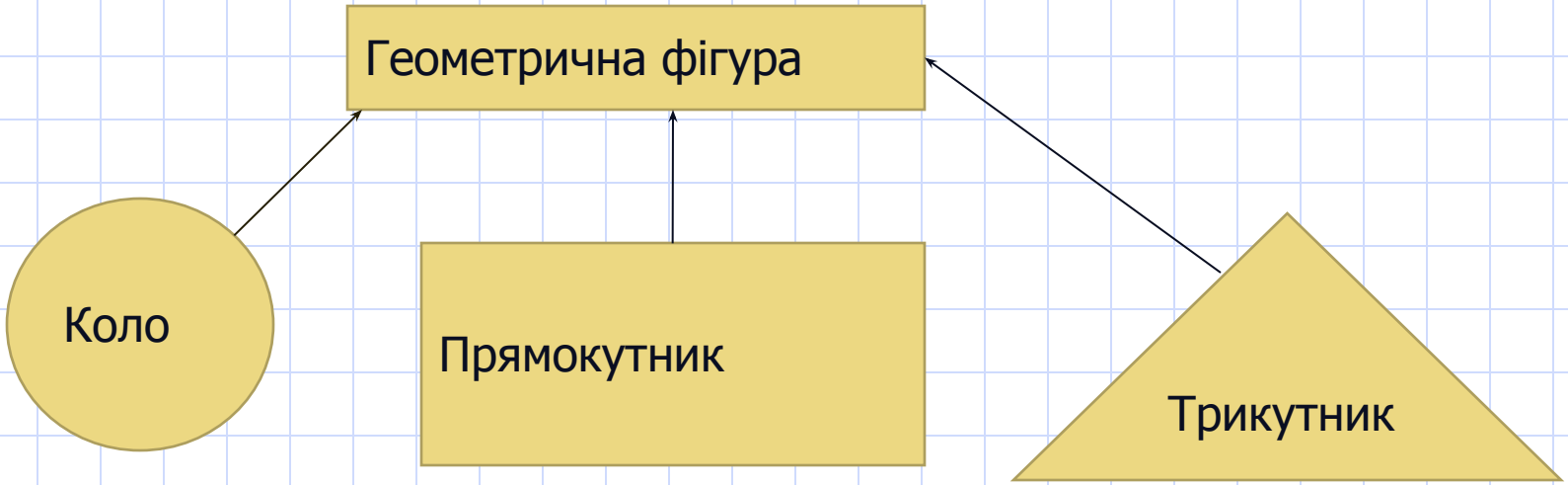
- **Пример 2**
  - связь между классом **Sale** и классом **CreditCardTransaction** (транзакция кредитной карточки)
  - каждая продажа соответствует ровно одному снятию денег с данной кредитной карточки
  - мощность - "один-к-одному"

Наследование

- Отношение между классами, при котором класс использует структуру или поведение другого (**одиночное наследование**) или других (**множественное наследование**) классов
- Вводит иерархию "общее/частное"
- Множественное наследование
  - в C++ - возожно, однако создает трудности!
  - в Java – запрещено, однако возможно множественная реализация интерфейсов

Підкласи

Підкласи — це класи, які успадковують усі "властивості" суперкласу ("батьківського класу")



Приклад

```
class PersonnelRecord
    {
       public:
         char * employee Name () const;  //всі об'єкти можуть
         int employee Local Security Number () const;
                                          //отримати дані
         char * employee Department () const;
       protected:
         char name [100];                 //лише підкласи
         int Social Security Number;      //можуть визначати
         char department [10];            //значення
         float Salary;
    };
```

Ієрархія класів

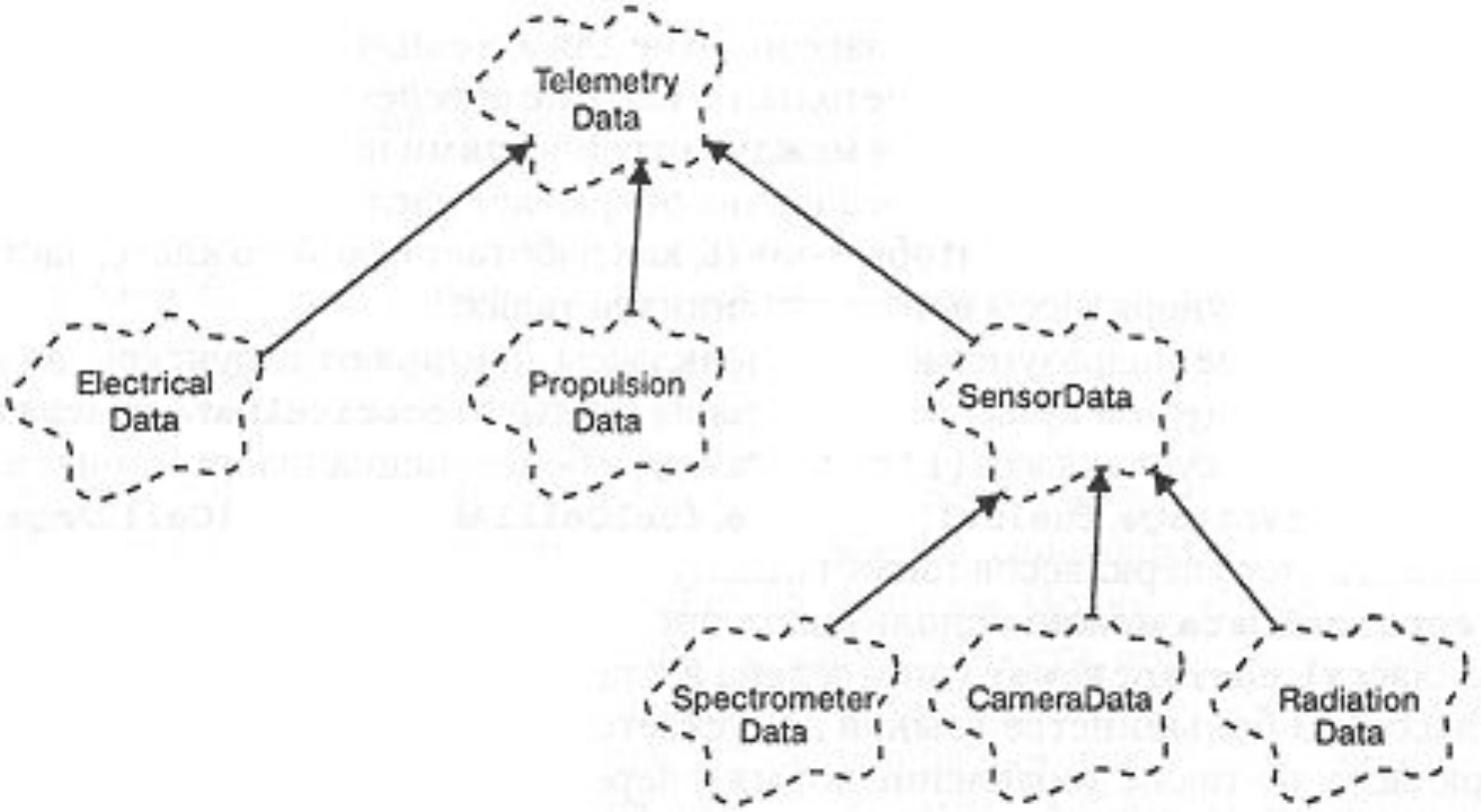
- Об’єкти різних класів і самі класи можуть перебувати у відношенні успадкування, за якого формується ієрархія об’єктів, що відповідає заздалегідь передбаченій ієрархії класів.
- Ієрархія класів дозволяє визначати нові класи на основі вже існуючих. Існуючі класи зазвичай називають ***базовими*** (інколи ***батьківським***), а нові класи, що формуються на основі базових, — ***похідними*** (***породженими***), інколи ***класами-нащадками*** або ***спадкоємцями***. Похідні класи “отримують спадок” — дані і методи своїх базових класів — і, крім того, можуть поповнюватись власними компонентами (даними і власними методами).
- Наприклад, за таким визначенням

  class S: X{…};

  клас S породжений класом X, звідки він успадковує компоненти.
- Віртуальні функції визначаються специфікатором virtual і дозволяють програмісту описати в базовому класі функції, які можна було б замінити у кожному наступному класі.

Успадкування

- **Приклад 1** – дані космічних апаратів



Успадкування

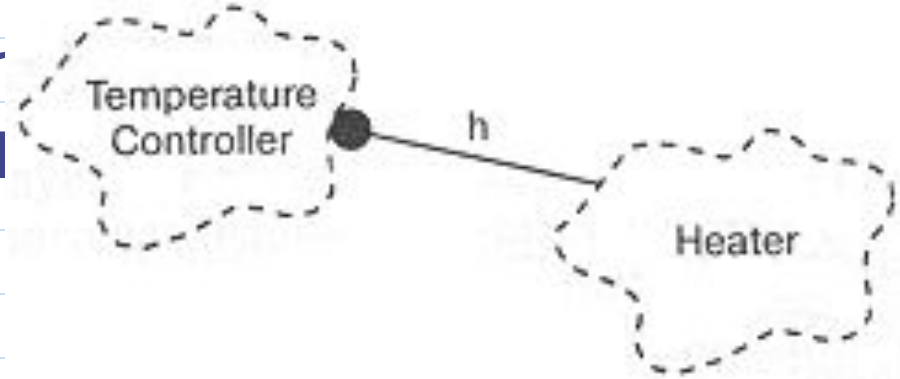
```
class TelemetryData {
public:
 TelemetryData();
 virtual ~TelemetryData();
 virtual void transmit();
 Time currentTime() const;
protected:
 int id;
 Time timeStamp;
};
```

Успадкування

```
class ElectricalData : public TelemetryData {
public:
 ElectricalData(float v1, float v2, float a1, float
a2);
 ~ElectricalData();
 void transmit();
 float currentPower() const;
protected:
 float fuelCell1Voltage, fuelCell2Voltage;
 float fuelCell1Amperes, fuelCell2Amperes;
};
```

Агрегація

- Виражає **відношення цілого та частини**
- ***Агрегація*** — включення одного класу до іншого — відповідає *агрегації* між екземплярами.
- Агрегація як співвідношення "*ціле–частина*" є *спрямованою*. Не вимагає *обов'язкового* фізи
(акціонер *володіє* а
складається з них).



- Якщо (і тільки якщо) існує відношення

Агрегація

```
class TemperatureController {
public:
 TemperatureController(Location);
 ~TemratureController();
 void process(const
TemperatureRamp&);
 Minute schedule(const
TemperatureRamp&) const;

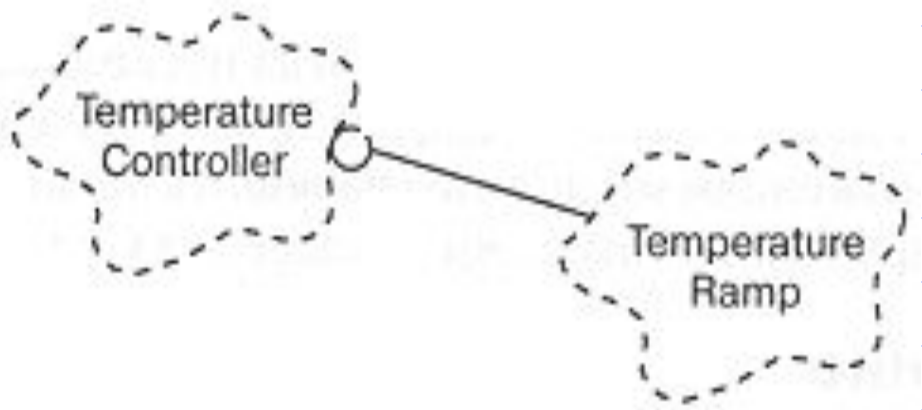
private:
 Heater h;
};
```

Агрегація

- Агрегація не вимагає **обов’язкового фізичного включенння об’єкта**!
- **Приклад 2**
  - акционер владеет акциями, но они не являются его физической частью
  - время жизни объектов может быть совершенно различным!

Використання

- ***Використання*** — відношення між класами, якщо клас є частиною сигнатури функції-члена іншого класу (параметром).
- Використання класів –> рівноправний зв'язок між їх екземплярами (*клієнт–сервер*).
- **Пример 1**
  - **TemperatureController** контролер температури
  - **TemperatureRamp** – визначає функцію часу від температури



Приклад використання

```
class TemperatureController {
public:
 TemperatureController(Location);
 ~TemratureController();
 void process(const TemperatureRamp&);
 Minute schedule(const TemperatureRamp&)
const;

private:
 Heater h;
};
```

Инстанцирование

- Подстановка параметров шаблона обобщенного или параметризованного класса
- В результате создается конкретный класс, который может иметь экземпляры
- В C++
  - использование параметризированных классов - шаблонов

Метакласс

- Класс классов. Класс, экземпляры которого сами являются классами
- В чистом виде нет в C++!
- Реализуется за счет использования **статических** (static) методов и атрибутов
  - они являются общими для всех экземпляров этого класса

Лекция 5

Идентичность объектов

Идентичность

- *Идентичность - это такое свойство объекта, которое отличает его от всех других объектов*
- «В большинстве языков программирования и управления базами данных для различения временных объектов их именуют, тем самым путая адресуемость и идентичность. Большинство баз данных различают постоянные

Пример

```
struct Point {
    int x;
    int y;
    Point() : x(0), y(0) {}
    Point(int xValue, int yValue) :
    x(xValue), y(yValue) {}
};
```

Если абстракция представляет собой

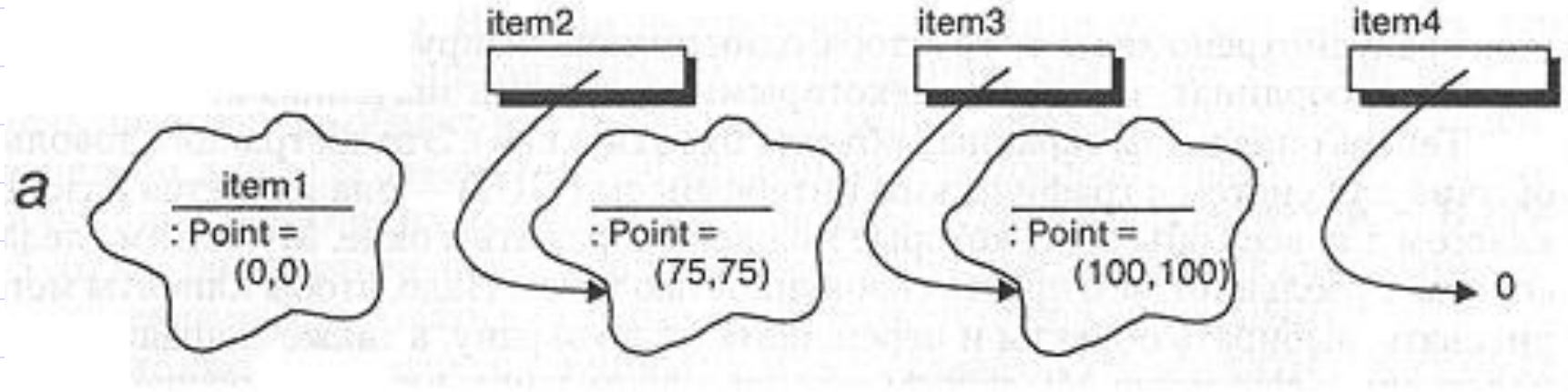
Пример

```
class DisplayItem {
public:
    DisplayItem();
    DisplayItem(const Point& location);
    virtual ~DisplayItem();
    virtual void draw();
    virtual void erase();
    virtual void select();
    virtual void unselect();
    virtual void move(const Point&
```

Экземпляры классов

- Объявим экземпляры указанных классов:

```
DisplayItem item1;
DisplayItem* item2 = new
DisplayItem(Point(75, 75));
DisplayItem* item3 = new
```

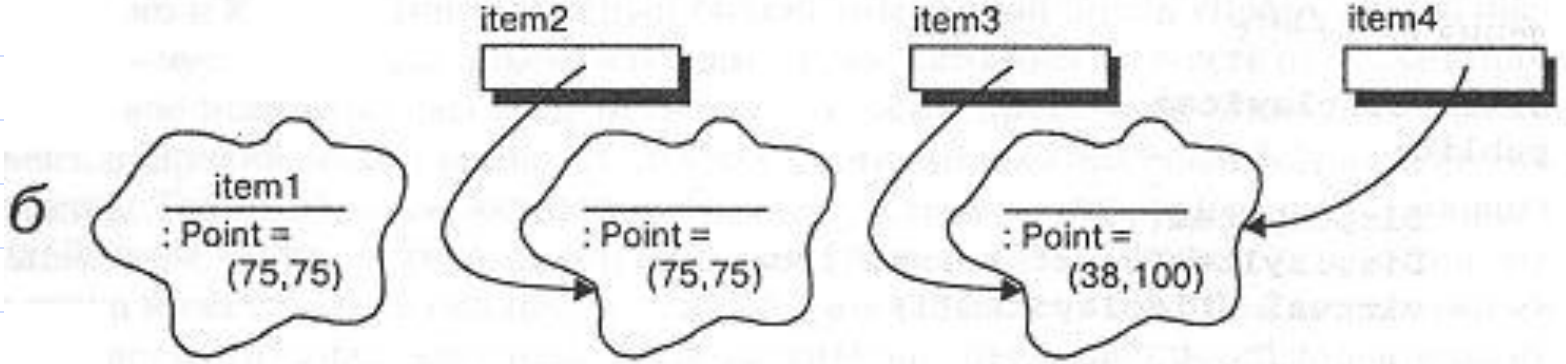


Идентичность

- Рассмотрим результат выполнения следующих операторов

```
item1.move(item2->location());
item4 = item3;
item4->move(Point(38, 100));
```

- Объект **item1** и объект, на который указывает **item2**, теперь



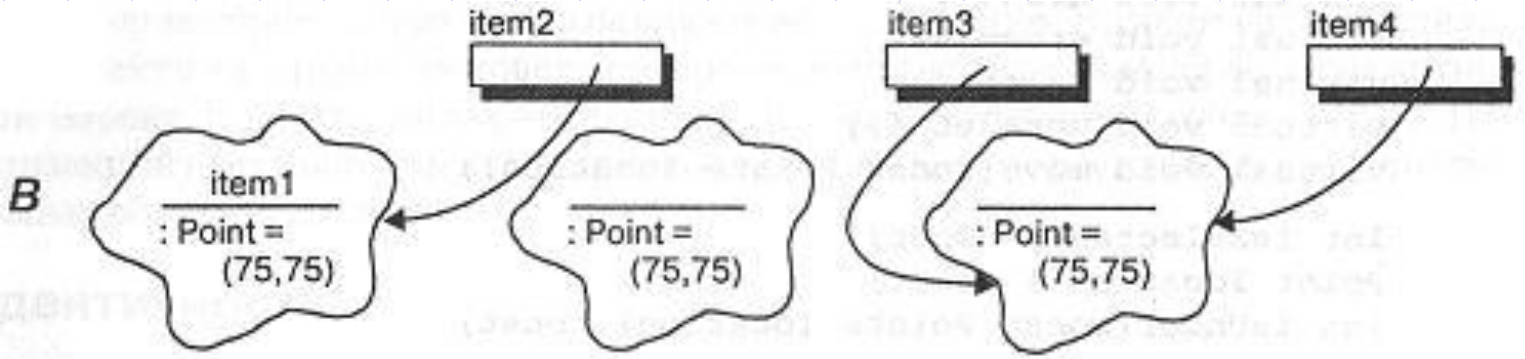
Идентичность

- Хотя объект **item1** и объект, на который указывает **item2**, имеют одинаковое состояние, они остаются разными объектами.
- Кроме того, мы изменили состояние объекта ***item3**, использовав его новое косвенное имя **item4**.
- Эта ситуация называется **структурной зависимостью.**
- Структурная зависимость порождает в

Идентичность

- На рисунке иллюстрируется результат выполнения следующих действий:

item2 = &item1;



Идентичность

- В первой строке создается синоним: **item2** указывает на тот же объект, что и **item1**.
- Во второй доступ к состоянию **item1** получен через этот новый синоним. К сожалению, при этом произошла утечка памяти, - объект, на который первоначально указывала ссылка **item2**, никак не именуется ни прямо, ни косвенно, и его идентичность потеряна.

Копирование, присваивание и равенство

- **Структурная зависимость** имеет место, когда объект имеет ***несколько имен***. В наиболее интересных приложениях объектно-ориентированного подхода использование синонимов просто неизбежно. Например, рассмотрим следующие две функции:

```
void highLight(DisplayItem& i);
void drag(DisplayItem i); //
Опасно
```

Копирование, присваивание и равенство

- В C++ различают передачу параметров по ссылке и по значению. Надо следить за этим, иначе можно нечаянно *изменить копию* объекта, ***желая изменить сам объект***. В общем случае, передача объектов по ссылке крайне желательна для сложных объектов, поскольку при этом копируется ссылка, а не состояние, и следовательно, достигается большая эффективность (за исключением тех случаев, когда передаваемое значение очень простое)

Присваивание, копирование и равенство

- Присваивание - это, вообще говоря, копирование
- Копирование
  - «глубокое» - копирование всех ссылок и указателей на другие обьекты
  - «поверхностное» - копирует только объект, используя конструктор копирования
- Равенство может быть:
  - 2 объекта обозначают 1 объект (item3 и item4)
  - Равенство состояний у разных объектов (item1 и item2)

Равенство

- В C++ нет предопределенного оператора равенства, поэтому мы должны определить равенство и неравенство, объявив эти операторы при описании:
- Мы предлагаем описывать оператор равенства как виртуальный (так как ожидаем, что подклассы могут переопределять его поведение), и описывать оператор неравенства как

Время жизни объекта

- **Началом** времени существования любого объекта является момент его создания (отведение участка памяти)
- **Окончанием** - возвращение отведенного участка памяти системе.

Явное создание объекта

- Объекты создаются явно или неявно
- Явно
  - При объявлении (как это было с **item1**): тогда объект размещается в стеке
  - Можно разместить объект, то есть выделить ему память из «кучи» (**item3**)
- В C++ в любом случае при этом вызывается конструктор, который выделяет известное ему количество правильно инициализированной

Неявное создание объекта

- Неявно
  - передача параметра по значению в C++ создает в стеке временную копию объекта.
  - создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него
- Объекты, созданные в стеке, уничтожаются при выходе из блока, в котором они были определены, но объекты, созданные в "куче"

Лекція № 6

Успадкування

Успадкування

- Об’єкти різних класів і самі класи можуть перебувати у відношенні успадкування, за якого формується ієрархія об’єктів, що відповідає заздалегідь передбаченій ієрархії класів.
- Ієрархія класів дозволяє визначати нові класи на основі вже існуючих. Існуючі класи зазвичай називають ***базовими*** (інколи ***породжуюими***), а нові класи, що формуються на основі базових, — ***похідними*** (***породженими***), інколи ***класами-нащадками*** або ***спадкоємцями***.
- Похідні класи “отримують спадок” — дані і методи своїх базових класів — і, крім того, можуть поповнюватись власними компонентами (даними і власними методами). Елементи, які успадковуються, не переміщуються в похідний клас, а залишаються в базових класах. Повідомлення, обробку якого не можуть виконати методи похідного класу, автоматично передається в базовий клас.

Успадкування

- Будь-який похідний клас може, в свою чергу, стати базовим для інших класів, і таким чином формується напрямлений граф ієрархії класів та об’єктів. В ієрархії похідний об’єкт успадковує дозволені для успадкування компоненти всіх базових об’єктів. Іншими словами, в об’єкта є можливість доступу до даних і методів усіх своїх базових класів.
- Успадкування в ієрархії класів може відображатись і у вигляді дерева, і у вигляді більш загального ***напрямленого ациклічного графу***. В C++ дозволяється ***множинне успадкування*** — можливість для деякого класу успадковувати компоненти кількох ніяк не зв’язаних між собою базових класів.

Приклад class TelemetryData

```
class TelemetryData {
  public:
    TelemetryData();
    virtual ~TelemetryData();
    virtual void transmit();  //передача даних
    Time currentTime() const;
  protected:
    int id;       //ключ для  ідентифікації даних
    Time timeStamp; //часові мітки
};
```

Приклад class ElectricalData

```
class ElectricalData: public TelemetryData {
  public:
    ElectricalData(float v1, float v2, float a1,
                   float a2);
    virtual ~ElectricalData();
    void transmit();
    float currentPower() const;
                   //розвив потужність
  protected:
    float fuelCell1 Voltage, fuelCell2 Voltage;
                   //напруга та струм
    float fuelCell1 Amperes, fuelCell2 Amperes;
                   // в  обох  електробатареях
};
```

Основні означення

- Клас, від якого успадковуються властивості, називається ***суперкласом***. Спадкоємець — ***підкласом***.
- Класи, екземпляри яких не створюються, називаються ***абстрактними***.
- Підкласи наповнюють їх змістом.
- Найзагальніший клас — базовий. (Буває декілька.)

Приклад : базовий клас

- Предположим, у вас есть базовый класс **employee**:

```
#include <iostream.h>
#include <string.h>
class employee
{
    public:
       employee(char *, char *, float);
       void show_employee(void);
    protected:
       char name[64];
       char position[64];
       float salary;
    };
```

- Далее предположим, требуется класс **manager**, который добавляет следующие элементы данных в класс **employee**:

```
float annual_bonus;
char company_car[64];
```

Защищенные элементы обеспечивают доступ и защиту

- Программа не может обратиться напрямую к частным элементам класса. Для обращения к частным элементам программа должна использовать интерфейсные функции, которые управляют доступом к этим элементам.
- Как вы, вероятно, заметили, наследование упрощает программирование в том случае, если производные классы могут обращаться к элементам базового класса с помощью оператора точки. В таких случаях ваши программы могут использовать защищенные элементы класса.
- Производный класс может обращаться к защищенным элементам базового класса напрямую, используя оператор точку. Однако оставшаяся часть вашей программы может обращаться к защищенным элементам только с помощью интерфейсных функций этого класса.
- Таким образом, защищенные элементы класса находятся между открытыми (доступными всей программе) и закрытыми (доступными только самому классу) элементами.

Функції-члени базового класу

```
employee::employee(char *name, char *position, float
    salary) // конструктор
{
        strcpy(employee::name, name);
        strcpy(employee::position, position);
        employee::salary = salary;
    }
void employee::show_employee(void)
{
        cout << "Имя: " << name << endl;
        cout << "Должность: " << position << endl;
        cout << "Оклад: $" << salary << endl;
    }
```

Похідний клас

```
class manager : public employee
{
   public:
      manager(char *, char *, char *, float,
   float);
      void show_manager(void);
   private:
      float annual_bonus;
      char company_car[64];
      };
```

Функції-члени похідного класу

```
manager::manager(char *name, char *position, char *company_car,
    float salary, float bonus) : employee(name, position, salary)
    // ініціалізатор конструктора
{
      strcpy(manager::company_car, company_car) ;
      manager::annual_bonus = bonus ;

    }
void manager::show_manager(void)
{
      show_employee(); // без явної кваліфікації (підклас є
          //підтипом)
      cout << "Машина фирмы: " << company_car << endl;
      cout << "Ежегодная премия: $" << annual_bonus << endl;

    }
```

Головна функція

```
void main(void)
{
    employee worker("Джон Дой", "Программист", 35000); // конструктор базового класу
    manager boss("Джейн Дой", "Вице-президент ", "Lexus", 50000.0, 5000); // конструктор похідного класу
    worker.show_employee() ;
    boss.show_manager();
}
```

Резюме

- Наследование – отношение между классами, позволяющее производить новый класс из существующего базового класса.
- Производный класс — это новый класс, а базовый класс — существующий класс.
- Для порождения класса из базового начинайте определение производного класса ключевым словом class, за которым следует имя класса, двоеточие и имя базового класса, например class dalmatian: dog.
- Когда вы порождаете класс из базового класса, производный класс может обращаться к общим элементам базового класса, ***как будто эти элементы определены внутри самого производного класса***. Для доступа к частным данным базового класса производный класс должен использовать **интерфейсные функции базового класса**.
- Внутри конструктора производного класса ваша программа должна вызвать конструктор базового класса, указывая двоеточие, имя конструктора базового класса и соответствующие параметры сразу же после заголовка конструктора производного класса.

Резюме

- Чтобы обеспечить **производным классам прямой доступ** к определенным элементам базового класса, в то же время защищая эти элементы от оставшейся части программы, C++ обеспечивает защищенные {protected) элементы класса. Производный класс может обращаться к защищенным элементам базового класса, как будто они являются общими. Однако для оставшейся части программы защищенные элементы эквивалентны частным.
- Если в производном и базовом классе есть элементы с одинаковым именем, то внутри функций производного класса C++ будет использовать элементы производного класса. Если функциям производного класса необходимо обратиться к элементу базового класса, вы должны использовать оператор глобального разрешения, например base class:: member.

Поліморфізм

- VCL — візуальні компоненти. TObject — базовий клас.
- Клас має 2 типи клієнтів:
  - – екземпляри — зовнішня візуальна поведінка (відкрита частина опису)
  - – підкласи — визначаються захищеною частиною опису класу.
- Поведінка класу успадковується. Функція, оголошена віртуальною, може бути перевизначена в підкласі, а інші — ні. В підкласах можуть бути додані нові функції.

Поліморфізм

- Розглянемо приклад
- Нехай функція transmit класу TelemetryData реалізована так:

Поліморфізм

- void TelemetryData::transmit()
  //передає заголовок пакету
  {
  //передати id
  //передати timeStamp
  };
  void ElectricalData::transmit()
  //перевизначення

  {
  TelemetryData::transmit ( );
  //виклик функції суперкласу з використанням
  //кваліфікованого імені
  //передати напругу
  //передати силу току
  };

Поліморфізм

- Визначимо екземпляри двох класів:

```
TelemetryData telemetry;
ElectricalData electrical (5.0, -5.0, 3.0, 7.0);
```

- Визначимо вільну процедуру:

```
void transmit FreshData(TelemetryData& d,
                        const Time& t)
{
  if(d, currentTime() >= t)
     d.transmit();
};
```

Поліморфізм

i transmit FreshData(telemetry, Time (60));
ii transmit FreshData(electrical, Time (120));

- i — перша функція передає заголовок, ii — передає заголовок та 4 дійсних числа. Це приклад поліморфізму.
- Змінна d може позначати об'єкти різних класів, в яких є загальний суперклас — параметричний поліморфізм.

Перенавантаження

- Вперше ідею поліморфізму пов'язували з можливістю перевизначати зміст символів, наприклад "+". В сучасному програмуванні це називається *перевантаженням*:
  - функцій (відрізняються сигнатурою)
  - операцій.
- За умов відсутності поліморфізму код програми має містити оператори switch та case.

Механізм пізнього зв’язування

- Поліморфізм тісно пов‘язаний з механізмом пізнього зв‘язування: зв ‘язок методу та імені за умов поліморфізму визначаються тільки в процесі виконання програми.
- Якщо функція віртуальна, то зв ‘язування пізнє, звідси функція поліморфна. Якщо ні — зв‘язок здійснюється під час компіляції.

Успадкування та типізація

- В діаграмі класів TelemetryData усі підкласи є підтипами вищого класу.
- Система типів, паралельна успадкуванню, характерна для мов з сильною типізацією, в тому числі C++.
- Паралель між типізацією та успадкуванням з'являється там, де ієрархія загального і часткового висловлює смислові зв'язки між абстракціями.

Приклад успадкування

- TelemetryData telemetry;
  ElectricalData electrical(5.0, -5.0, 3.0, 7.0);
  telemetry = electrical; //правомірно, так як
  //electrical — підтип telemetry
- Проте таке присвоєння небезпечне, оскільки втрачаються усі доповнення в стані підкласу, якщо порівнювати із станом суперкласу.
- electrical = telemetry; //невірно

Висновок з прикладу

- Висновок: присвоєння об'єкту y значення об 'єкту x (y=x) припустимо, якщо тип об'єкту x співпадає з типом об'єкту y, або x є його підтипом.
- Для типів, для яких існує співвідношення клас/підклас, припустиме перетворення типів (в C++ — зведення типів). Звичайно зведення типів використовують для присвоєння об'єкту спеціалізованого класу об'єкта більш загального класу. Іноді навпаки (це небезпечно).

Ієрархія типів

- В C++ ієрархія типів співпадає з ієрархією класів. Таким чином, техніку виклику методів суперкласу з підкласу можна оптимізувати.
- Якщо під час визначення класу його суперклас оголошено public, то підклас є одночасно підтипом (як в TelemetryData): він зобов'язується виконувати усі обов'язки суперкласу:
  - забезпечує сумісну з суперкласом підмножину інтерфейсу
  - має ідентичну з суперкласом поведінку.

Private у суперкласі

- Якщо під час оголошення класу його суперклас оголосити private, то підклас спадкує структуру та поведінку, але не буде підтипом:
  - відкриті та захищені члени суперкласу стануть закритими членами підкласу, звідси вони будуть недосяжні для підкласів більш низького рівня.
  - підкласу та суперкласу будуть властиві несумісні (взагалі кажучи) інтерфейси з точки зору клієнта.

Приклад

```
class InternalElectricalData:
                        private ElectricalData
{
  public:
    InternalElectricalData(float v1, float
v2, float a1, float a2);
    virtual ~InternalElectricalData();
    ElectricalData::currentPower();
};
```

Висновок з прикладу

- В цьому прикладі суперклас закритий, звідси:

  1. його методи недосяжні для клієнтів;

  2. об'єкти підкласу не можна присвоювати об'єктам суперкласу;

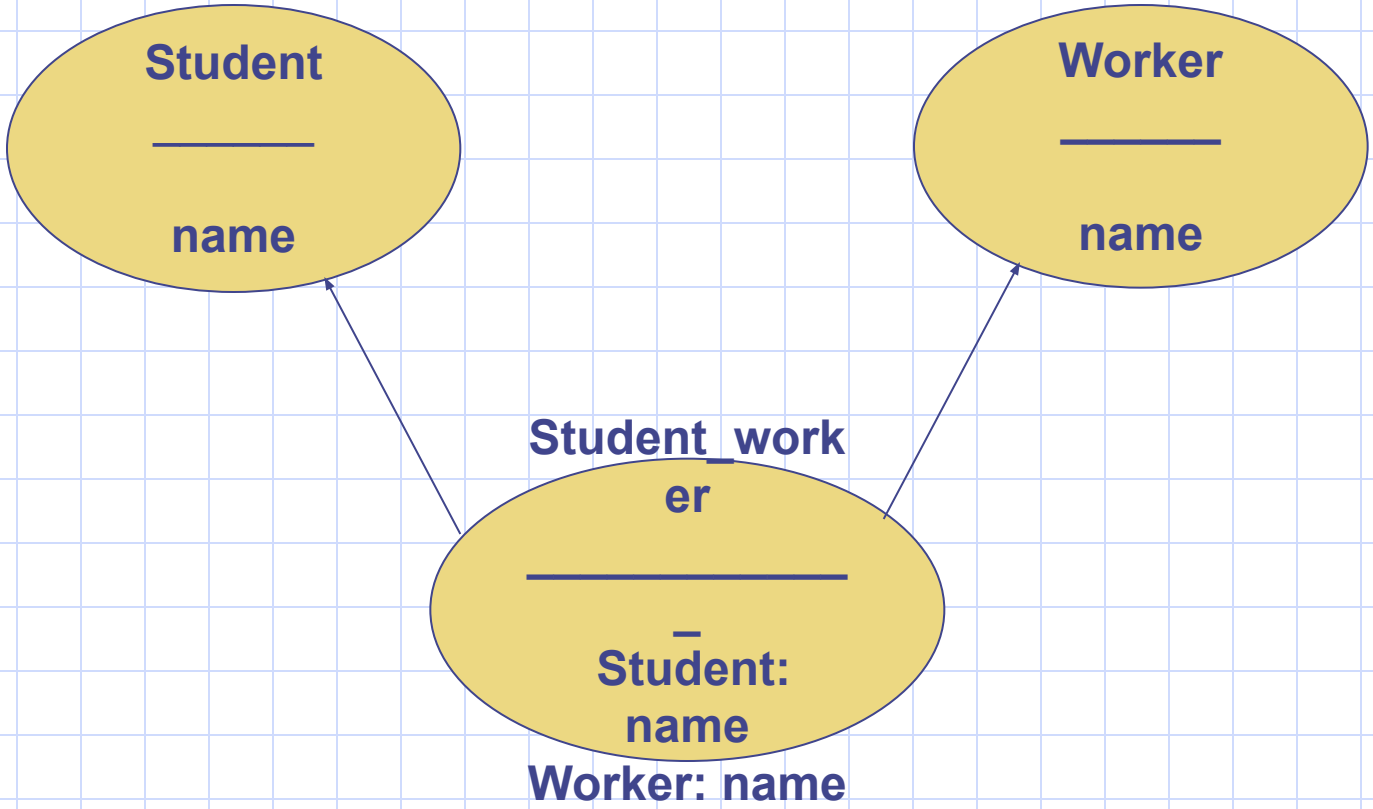
  3. функція currentPower() видима завдяки її явній кваліфікації (інакше вона була б недосяжною).

- Наслідуваний елемент не можна робити більш відкритим в підкласі, ніж в суперкласі.
- Наприклад, член timeStamp (protected в TelemetryData) не може стати public шляхом явної кваліфікації.

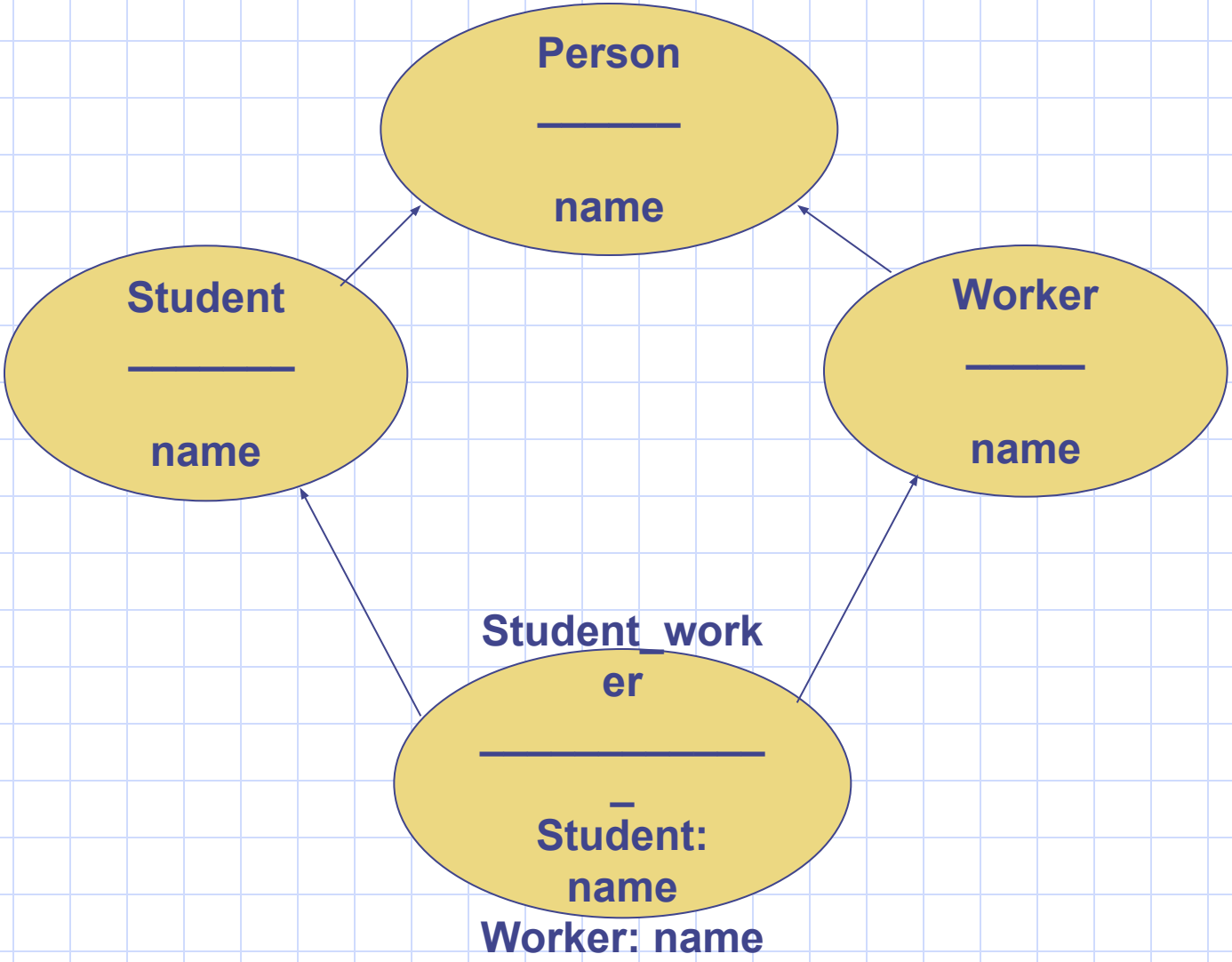
Множинне успадкування

- **Розглянемо приклад класу Student**
- class student: public virtual person {
};
class student {
  public:
    char* name;
};
class worker {
  public:
    char* name;
};
- class student_worker: public student,
                        public worker
{
  void worker() {
- print() { cout<<name};
};

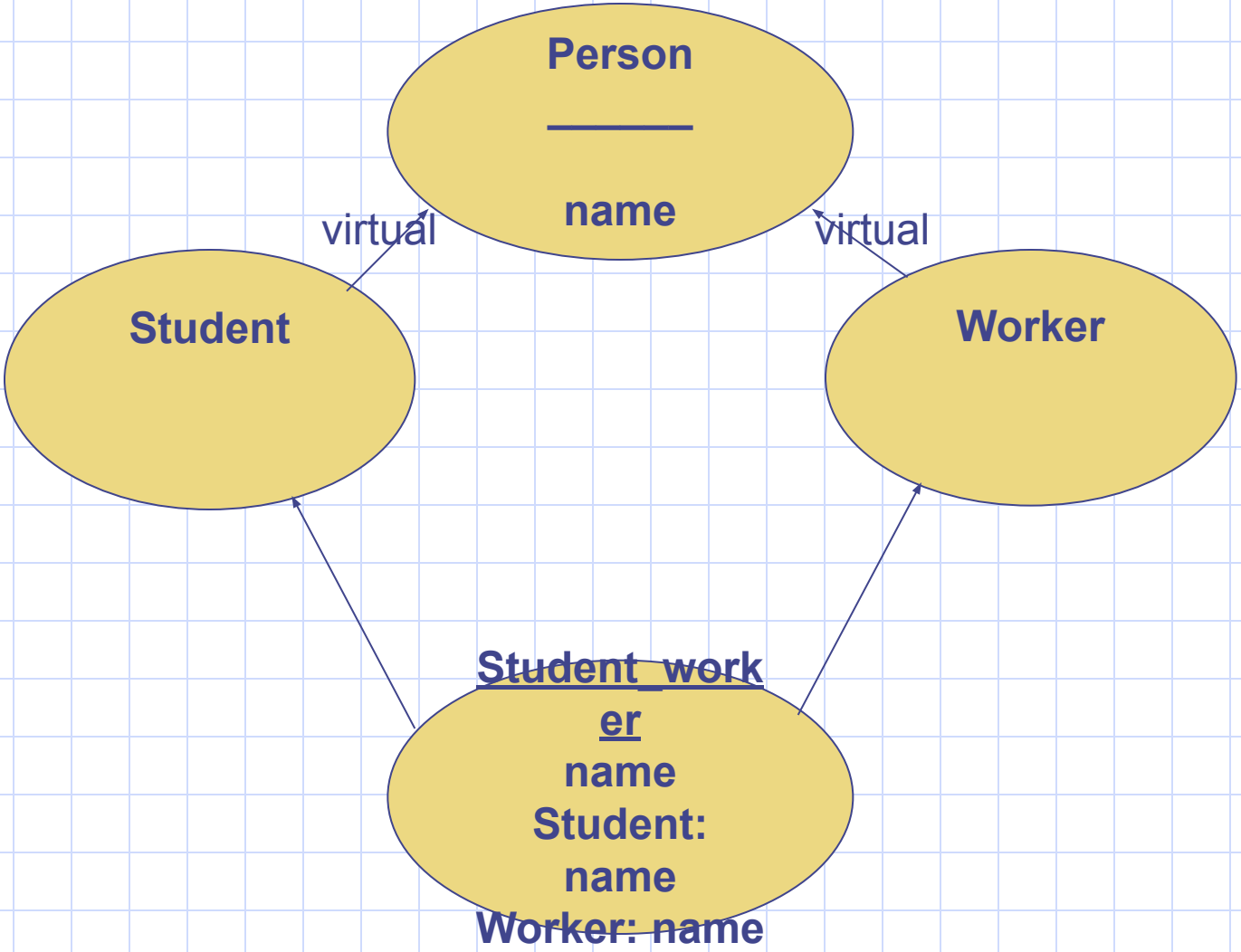
Проблеми множинного успадкування



Проблеми множинного успадкування



Проблеми множинного успадкування



Проблеми множинного успадкування

- *конфлікт імен між суперкласами* в C++ усувається шляхом додавання префікса (повної кваліфікації)
- *повторне успадкування*: один клас є спадкоємцем іншого за кількома лініями
  - можна заборонити (Smalltalk)
  - можна розвести дві копії успадкованого елемента, додаючи до імен префікси у вигляді імені класу-джерела (C++)
  - множинні посилання на один і той же клас можна розглядати як один клас (C++); суперклас, що повторюється, визначається як віртуальний базовий клас (shared class)

Множинне успадкування

- При множинному успадкуванні використовується прийом створення *домішок* (mixin) — класів, не призначених для породження самостійних примірників, а для змішування з іншими класами (InsurableItem, InterestBearingItem).
- *Домішка* — це клас, який виражає не поведінку, а одну звичку.
- Класи, сконструйовані з домішок, називаються *агрегатними*.
- Множинне успадкування використовується, якщо існує декілька ортогональних наборів ознак, за якими можна згрупувати кінцеві класи і ці групи перекриваються.

Відношення між класами та об’єктами

- Класи і об'єкти тісно пов'язані поняття. Будь-який об'єкт належить деякому класу. Клас породжує будь-яке число об'єктів.
- Класи статичні (всі їх особливості і зміст визначаються в процесі компіляції). Об'єкти динамічні, тобто створюються і знищуються в процесі виконання програми.

Відношення між класами та об’єктами

- На етапі аналізу і ранніх стадіях проектування вирішується дві основні задачі:
  - виявлення класів і об'єктів, що складають словник предметної області
  - побудова структур, що забезпечують взаємодію об'єктів, при яких виконуються вимоги задачі

Якість об’єктів

- Якість класів і об'єктів визначається критеріями:
  - зчеплення - міра глибини зв'язків між окремими модулями або класами
  - суперечність: треба прагнути до мінімального зачеплення, але успадкування передбачає сильне зачеплення

Якість класів

- ***Зачем нужны виртуальные функции***
  При наследовании часто бывает необходимо, чтобы поведение некоторых методов базового класса и классов-наследников отличались. Решение, на первый взгляд, очевидное: переопределить соответствующие методы в производном классе. Однако тут возникает одна проблема, которую лучше рассмотреть на простом примере (листинг 9.1).

```
//Листинг 9.1. Необходимость виртуальных функций
#include <iostream> using namespace std;
class Base // базовый класс
{ public:
int f(const int &d) // метод базового класса
{ return 2*d; }
int CallFunction(const int &d) // предполагается
{ return f(d)+1; // вызов метода базового класса
}
};
class Derived: public Base // производный класс
{ public: // CallFunction наследуется
int f(const int &d) // метод f переопределяется
 { return d*d;
}
};
int main()
{
Base a; // объект базового класса
cout << a.CallFunction(5)<< endl; // получаем 11
Derived b; // объект производного власса
cout << b.CallFunction(5)<< endl; // какой метод f вызывается?
return 0;
 }
```

- В базовом классе определены два метода — **f()** и **CallFunction()**, — причем во втором методе вызывается первый. В классе-наследнике метод **f()** переопределен, а метод **CallFunction()** унаследован. Очевидно, метод **f()** переопределяется для того, чтобы объекты базового класса и класса-наследника вели себя по-разному. Объявляя объект b типа Derived, программист, естественно, ожидает получить результат **5 * 5 + 1 = 26** — для этого и переопределялся метод **f()**. Однако на экран, как и для объекта a типа Base, выводится число 11, которое очевидно вычисляется как **2 * 5 + 1 = 11**. Несмотря на переопределение метода **f()** в классе-наследнике, в унаследованной функции **CallFunction()** вызывается «родная» функция **f()**, определенная в базовом классе!
  Аналогичная проблема возникает и в несколько другом контексте: при подстановке ссылки или указателя на объект производного класса вместо ссылки или указателя на объект базового. Рассмотрим опять пример с часами и будильником (листинг 9.2).

```
//Листинг 9.2. Неожиданная работа принципа подстановки
class Clock // базовый класс — часы
{ public:
        void print() const
        { cout << "Clock!" << endl; }
};
class Alarm: public Clock // производный класс — будильник
{ public:
        void print() const // переопределенный метод
        { cout << "Alarm!" << endl; }
};
void settime(Clock &d) // функция установки времени
{ d.print(); } // предполагается вызов метода базового класса
//...
Clock W; // объект базового класса
settime(W); // выводится "Clock«
Alarm U; // объект производного класса
settime(U); // ссылка на производный вместо базового
Clock *c1 = &W; // адрес объекта базового класса
c1->print(); // вызов базового метода
c1 = &U; // адрес объекта производного типа вместо базового
c1->print(); // какой метод вызывается, базовый или производный?
```

Опять в классе-наследнике переопределен метод для того, чтобы обеспечить различное поведение объектов базового и производного классов. Однако и при передаче параметра по ссылке базового класса в функцию settime(), и при явном вызове метода print() через указатель базового класса наблюдается одна и та же картина: всегда вызывается метод базового класса, хотя намерения программиста состоят в том, чтобы вызвать метод производного.
Для того чтобы разобраться в ситуации, необходимо уяснить, что такое связывание. Связывание — это сопоставление вызова функции с телом. В приведенных ранее примерах связывание выполняется на этапе трансляции (до запуска) программы. Такое связывание обычно называют ранним, или статическим.

При трансляции класса Base (см. листинг 9.1) компилятор ничего не знает о классах-наследниках , поэтому он не может предполагать, что метод f() будет переопределен в классе Derived. Его естественное поведение — «прочно» связать вызов f() с телом метода класса Base. Аналогично при трансляции функции settime() компилятору ничего не известно о типе реально передаваемого объекта во время выполнения программы. Поэтому вызов метода print() связывается с телом метода базового класса Clock, как и определено в заголовке функции settime(). Точно так же указатель на базовый класс «прочно» связывается с методом базового класса во время трансляции.
Конечно, при вызове метода по указателю в данном конкретном случае мы можем вызвать метод производного класса, задав явное преобразование указателя:

- static_cast<Alarm*>(c1)->print();
  Или так:
  ((Alarm *)c1)->print(); // "лишние" скобки нужны!
- Однако для функции settime() и метода CallFunction() это сделать невозможно — нам необходимо именно разное поведение в зависимости от типа объекта. Да и с указателем не все так просто: если такой вызов прописан внутри функции, которая принимает этот указатель как параметр (например, settime(Clock *c1)), то мы имеем те же проблемы.
  Определение виртуальных функций
  Получается, что в C++ должен существовать механизм, с помощью которого можно узнать тип объекта во время выполнения программы. Такой механизм в C++ есть и он, как уже отмечалось, называется динамической идентификацией типов (RTTI). Однако в ситуациях, подобных описанным, применяется другой, более «сильный» и элегантный механизм C++ — механизм виртуальных функций (см. п. 10.3 в Стандарте).
  Чтобы добиться разного поведения в зависимости от типа, необходимо объявить функцию-метод виртуальной; в C++ это делается с помощью ключевого слова virtual. Таким образом, в листинге 9.1 объявление метода f() в базовом и производном классе должно быть таким:

- virtual int f(const int &d) // в базовом классе
- { return 2*d; }
- virtual int f(const int &d) // в производном классе
- { return d*d; }

- После этого для объектов базового и производного классов мы получаем разные результаты: 11 и 26.
  Аналогично в листинге 9.2 объявление метода print() тоже должно начинаться со слова virtual:

- virtual void print() const // в базовом классе
- { cout << "Clock!" << endl; }
- virtual void print() const // в производном классе
- { cout << "Alarm!" << endl; }

- После этого вызов settime() с параметром базового класса обеспечит нам вывод на экран слова «Clock», а с параметром

- Для виртуальных функций обеспечивается не статическое, а динамическое (позднее, отложенное) связывание, которое реализуется во время выполнения программы. Естественно, это влечет за собой некоторые накладные расходы, однако на них можно не обращать внимания, так как обеспечивается динамический полиморфизм. Александреску указывает, что в C++ реализованы два типа полиморфизма:

- статический полиморфизм, или полиморфизм времени компиляции (compile-time polymorphism), осуществляется за счет перегрузки и шаблонов функций;

- динамический полиморфизм, или полиморфизм времени выполнения (run-time polymorphism), реализуется виртуальными функциями.
- С перегрузкой функций «разбирается» компилятор, правильно подбирая вариант функции в той или иной ситуации. И полиморфизм шаблонных функций тоже реализуется на этапе компиляции. Естественно, выбор осуществляется статически. Выбор же виртуальной функции происходит динамически — при выполнении программы. Класс, включающий виртуальные функции, называется полиморфным.
  Правила описания и использования виртуальных функций-методов следующие:
  1. Виртуальная функция может быть только методом класса.
  2. Любую перегружаемую операцию-метод класса можно сделать виртуальной, например, операцию присваивания или операцию преобразования типа.
  3. Виртуальная функция, как и сама виртуальность, наследуется.
  4. Виртуальная функция может быть константной.
  5. Если в базовом классе определена виртуальная функция, то метод производного класса с такими же именем и прототипом (включая тип возвращаемого значения и константность метода) автоматически является виртуальным (слово virtual указывать необязательно) и замещает функцию-метод базового класса.
  6. Конструкторы не могут быть виртуальными.
  7. Статические методы не могут быть виртуальными.
  8. Деструкторы могут (чаще — должны) быть виртуальными — это гарантирует корректный возврат памяти через указатель базового класса.

Лекція № 7

Система позначень Буча.

Мова UML

Системи позначень

- Загальноприйнята система позначень дозволяє розробнтку описати сценарій або розробити архітектуру та дохідчиво викласти свої решення колегам
- Чітка система позначень дозволяє автоматизувати більшу частину процесу перевірки на повноту та коректність
- Є засобом виразити результат роздумів над архітектурою та поведінкою системи, і не є самоціллю

Об’єктні моделі

- Статичні діаграми
  - класів
  - об’єктів
  - модулів
  - процесів
- Динамічні діаграми
  - переходів з одного стану в інше
  - взаємодії (між об’єктами)

Неможливо представити всі класи та їх зв’язки на одній діаграмі!

Об’єктні моделі

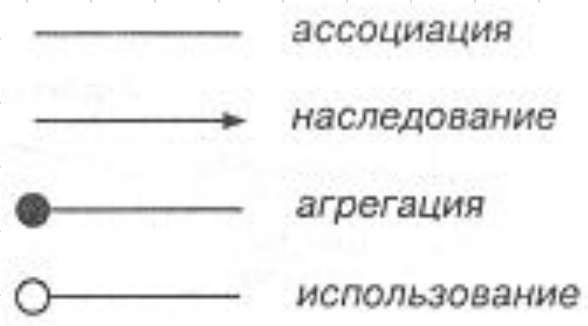
- Логічне представлення
  - описує перелік та зміст ключових абстракцій та механізмів, які формують предметну область або визначають архітектуру системи
- Фізична модель
  - визначає конкретну програмно-апаратну платформу, на якій реалізована система

Системи позначень

- Система позначень Буча
  - Нотації Буча
- Мова Unified Modeling Language
  - Уніфікована мова моделювання

Діаграми класів

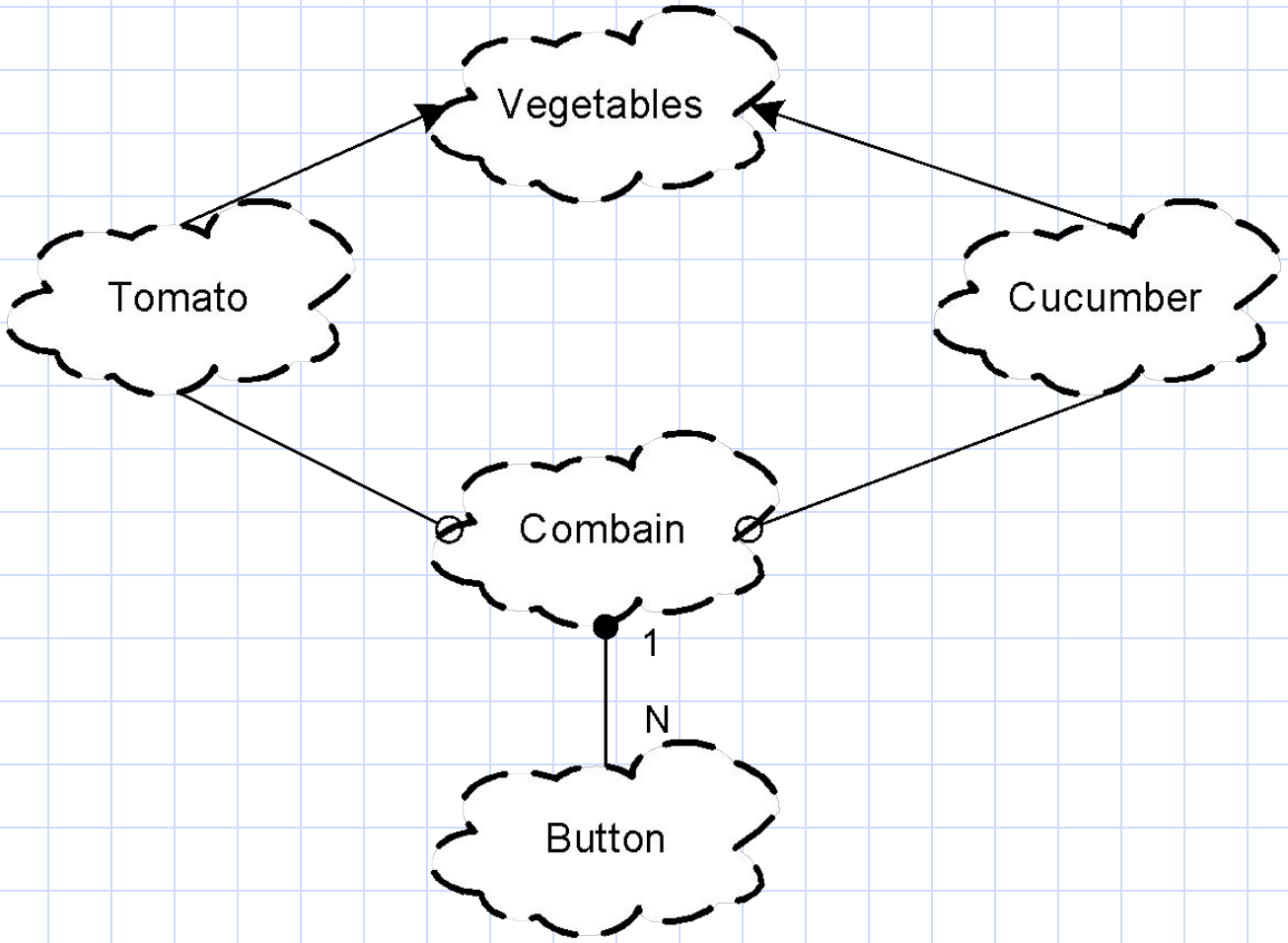
- Показують класи та їх відношення
  - асоціація
    - позначає деякий зв’язок між класами
  - успадкування
    - загальне–частинне
  - агрегація
    - ціле–частина
  - використання
    - наявність зв‘язку між екземплярами класів



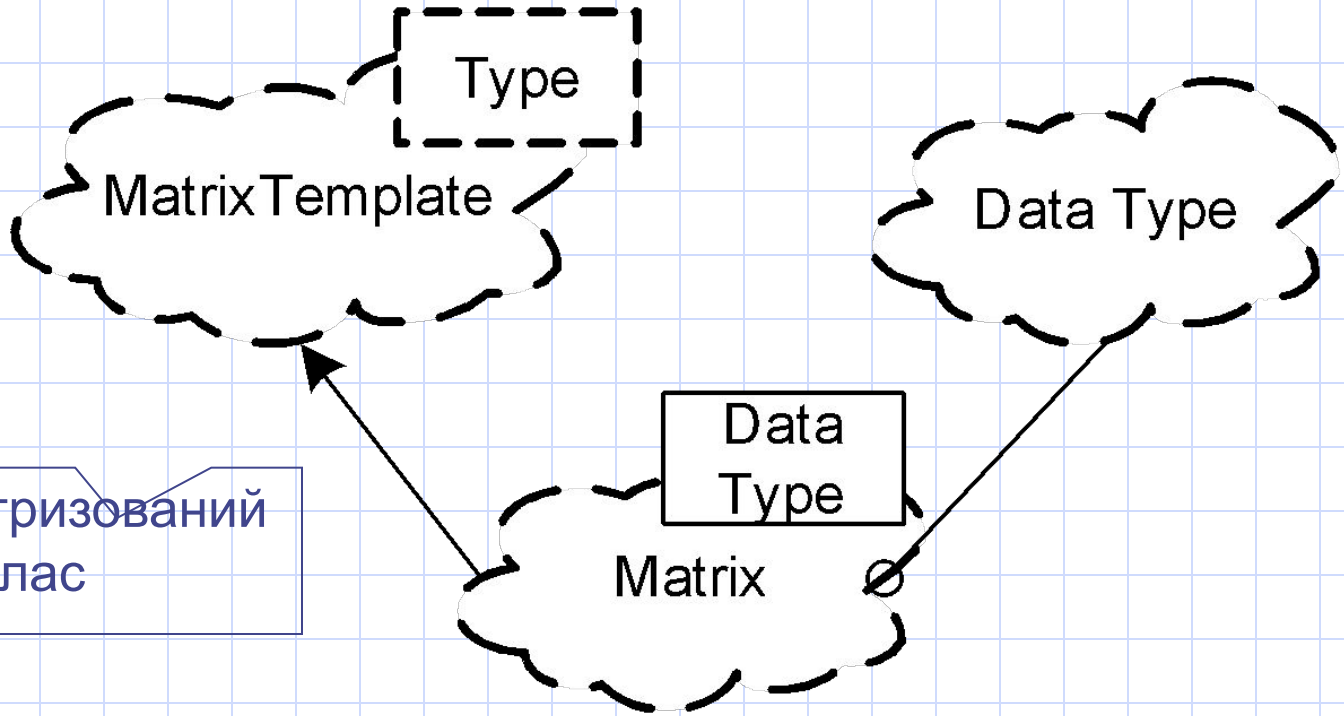
Потужність відношення

- **1** – один зв’язок
- **N** – необмежене число (0 або більше)
- **0..N** - Нуль або більше
- **1..N** - Один або більше
- **0..1** - Нуль або один
- **3..7** - Заданий інтервал
- **1..3, 7** - Заданий інтервал або точне число

Приклад діаграми класів



Відношення інстанціювання



Параметризований клас

Додаткові позначення

Абстрактний клас

Дружній клас

Статичний клас

Режими доступу

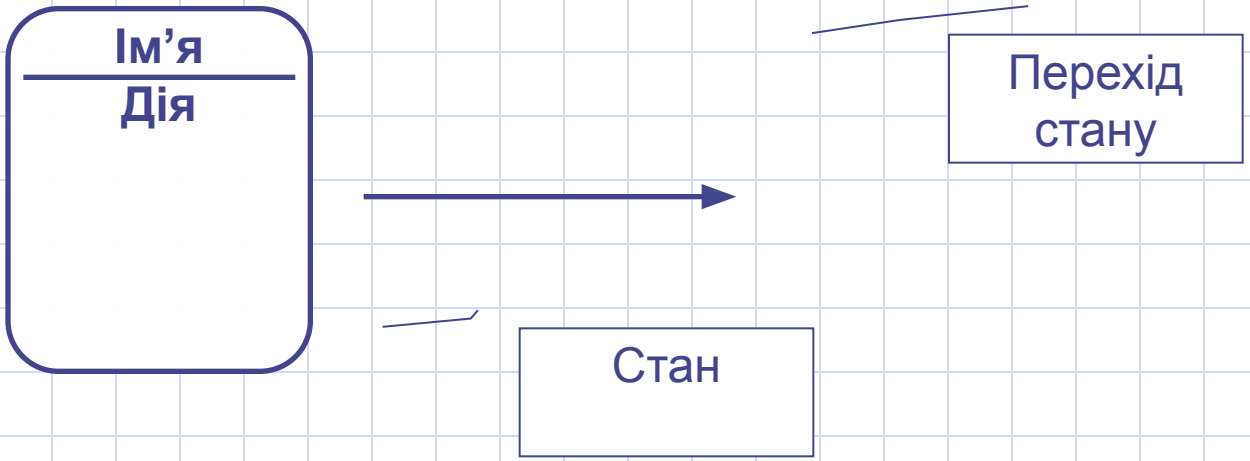
- **<без позначень>** - відкритий (за замовчуванням)
- **|** - protected
- **| |** - private

Специфікації

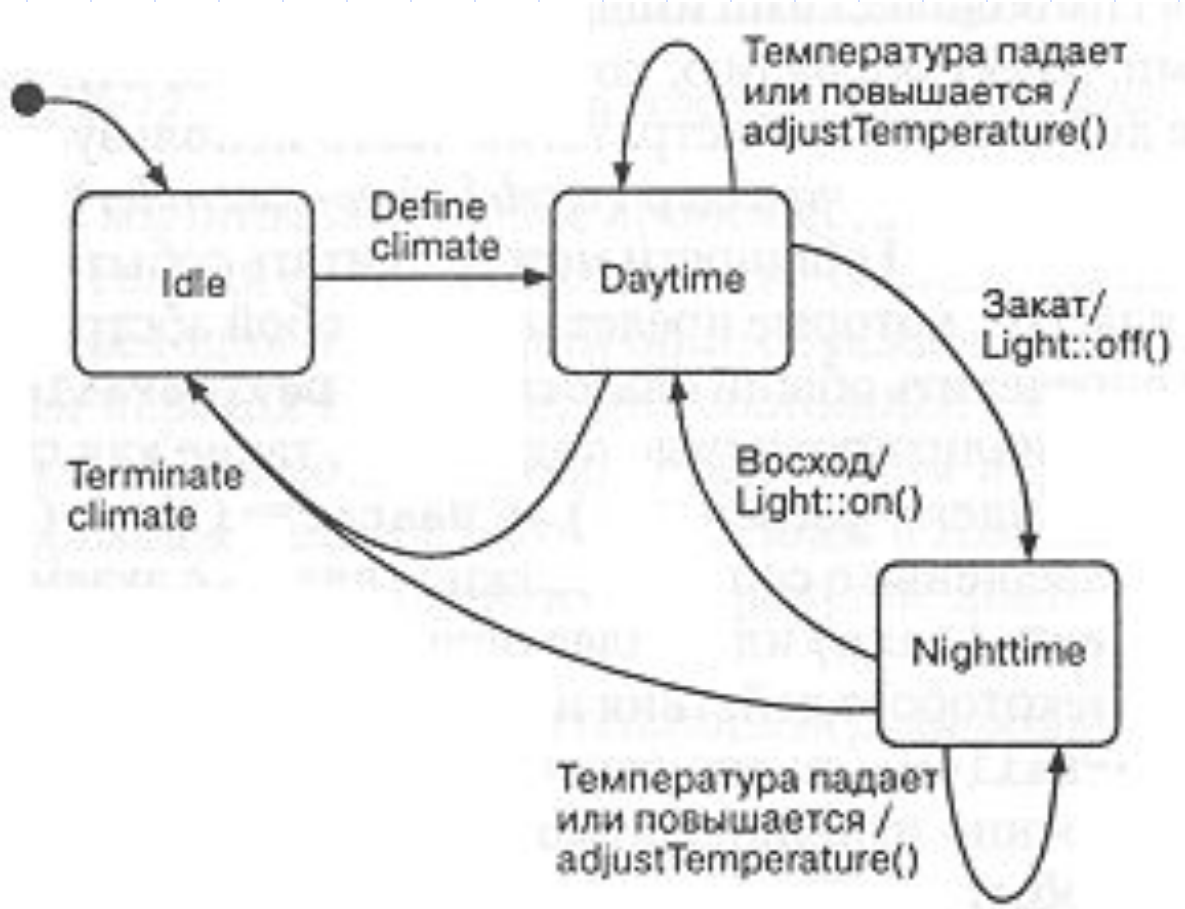
- Неграфічна форма, яка використовується для повного опису елементів системи позначень: класу, асоціацій, окремих операцій або цілої діаграми
- Наприклад
  - **Обов’язки: текст**
    Атрибути: перелік атрибутів
    Операції: перелік операцій
    Обмеження: перелік обмежень

Діаграми станів і переходів

- Відображає
  - простір станів даного класу
  - події, які спричиняють перехід з одного стану в інше;
  - дії, які відбуваються при зміні стану

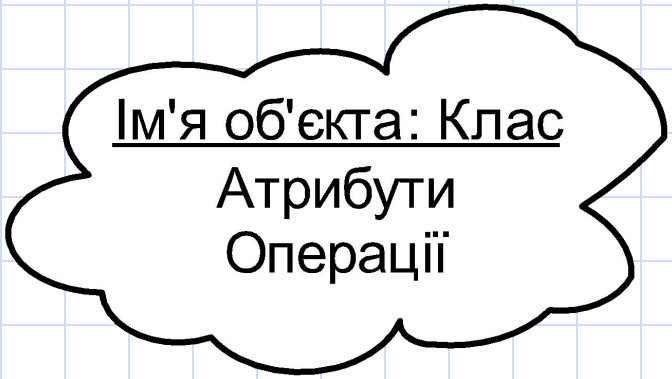


Приклад діаграми станів

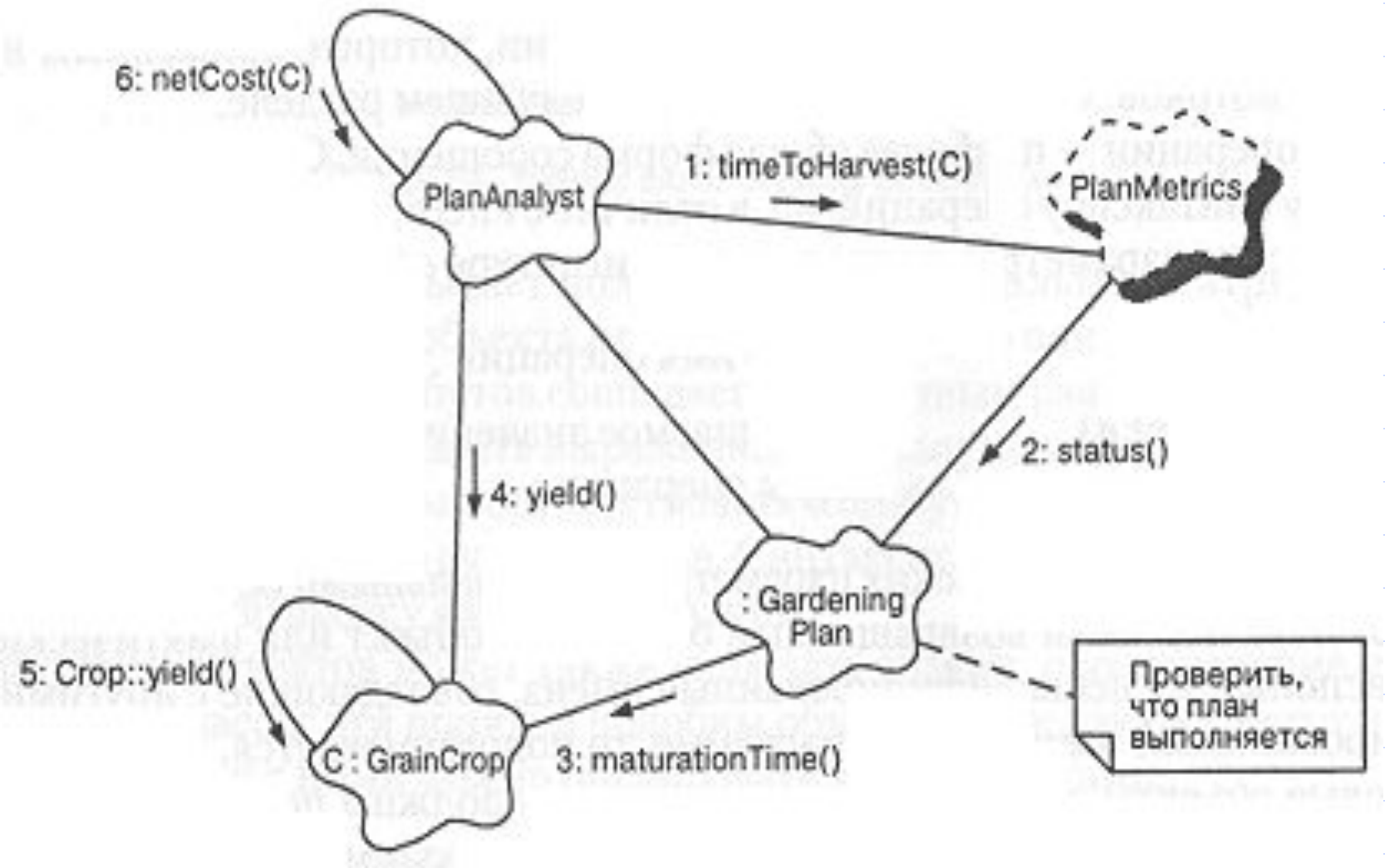


Діаграма об’єктів

- Відображає існуючі об’єкти та їх відношення в системі (миттєвий знімок роботи системи)

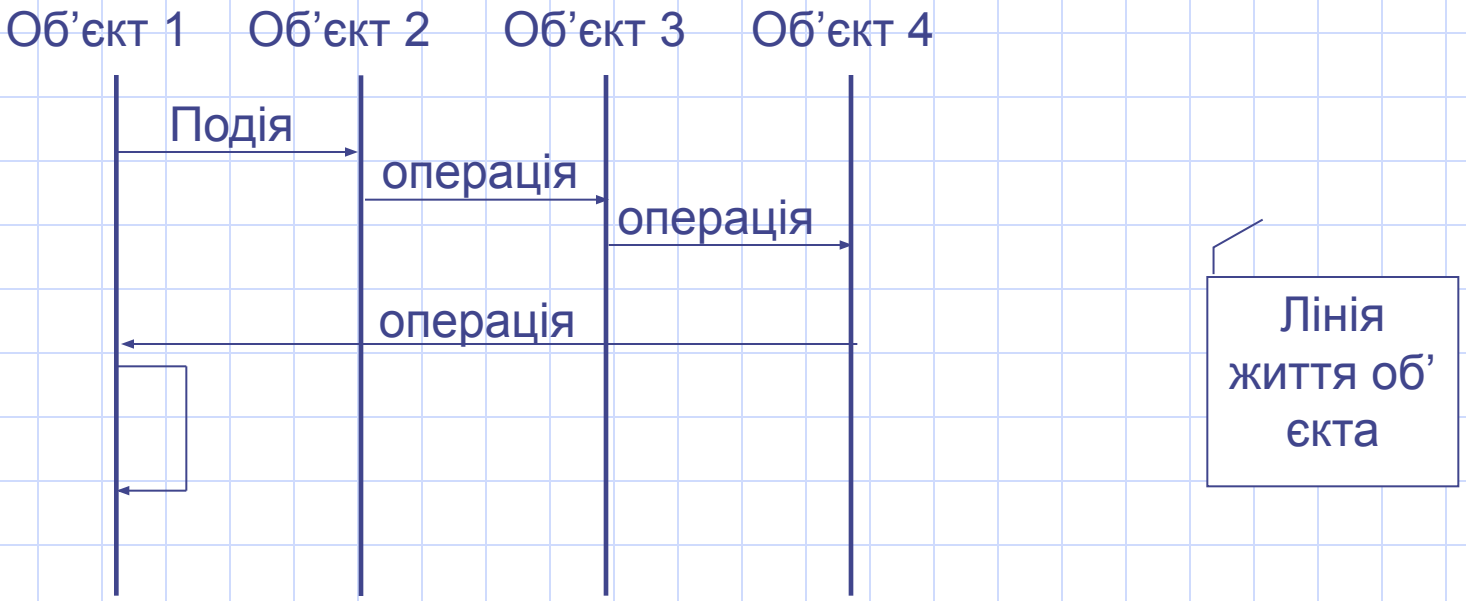


Приклад діаграми об’єктів

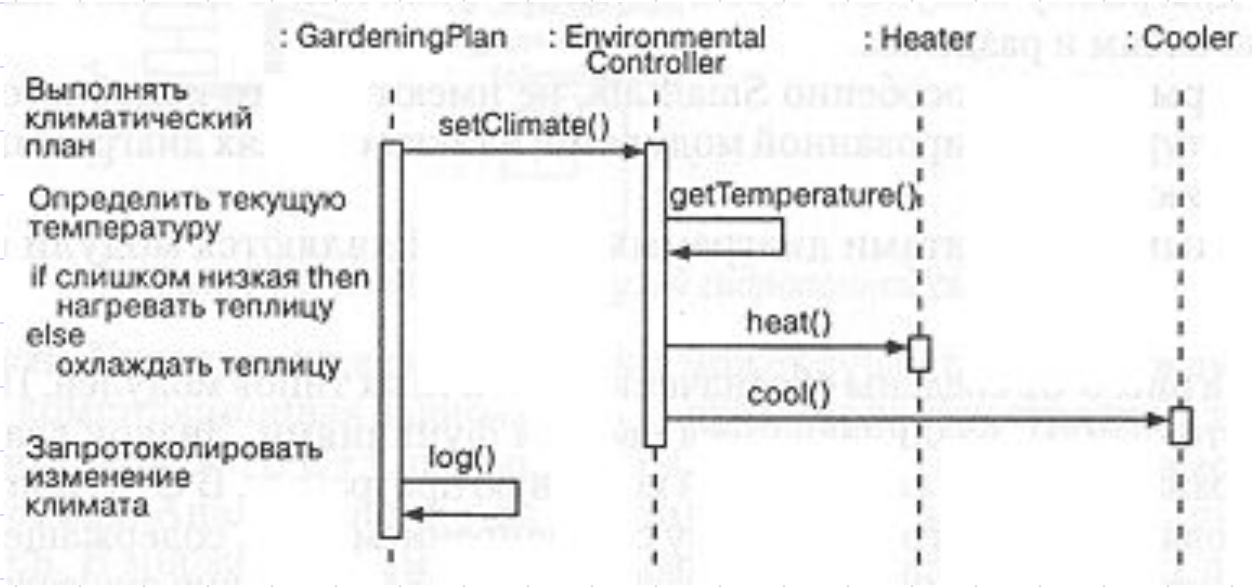


Діаграма взаємодії

- Використовується з тим, щоб прослідити виконання сценарія
- Інший спосіб представлення діаграми об’єктів

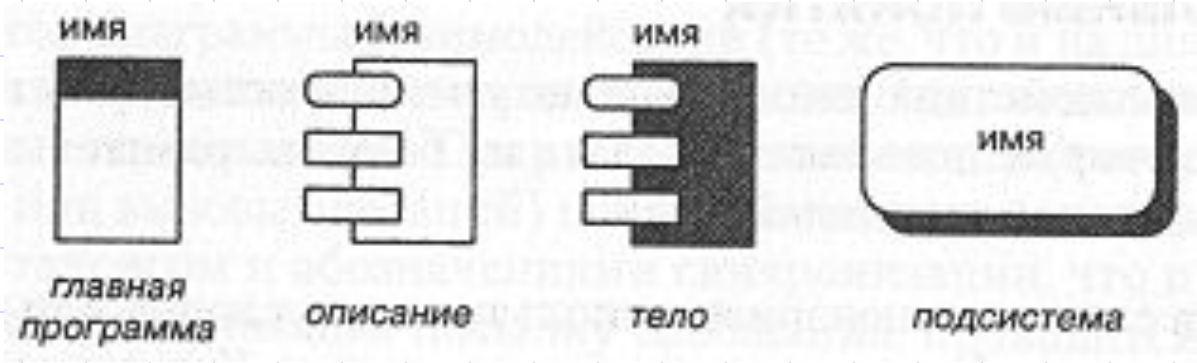


Приклад діаграми взаємодії



Діаграма модулів

- Відображає розподілення класів та об’єктів по модулям в фізичному проектуванні системи

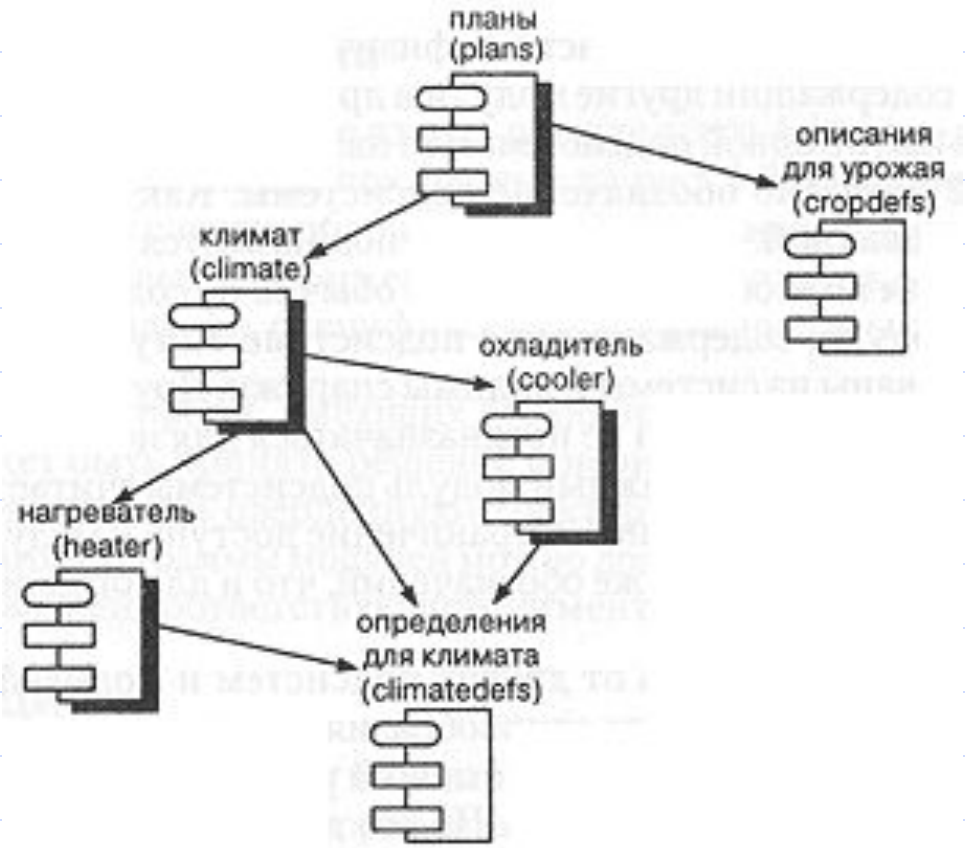


main()

.hpp

.cpp

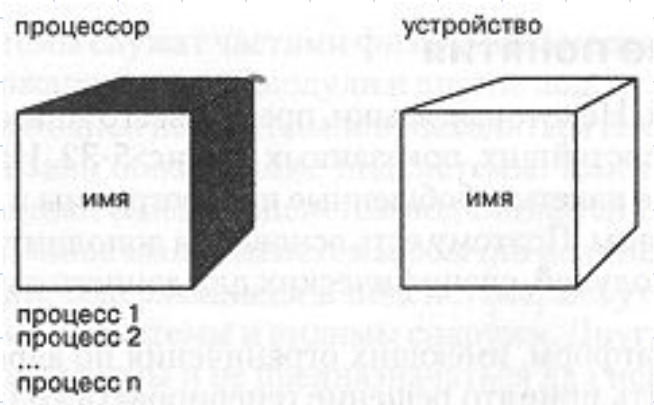
Приклад



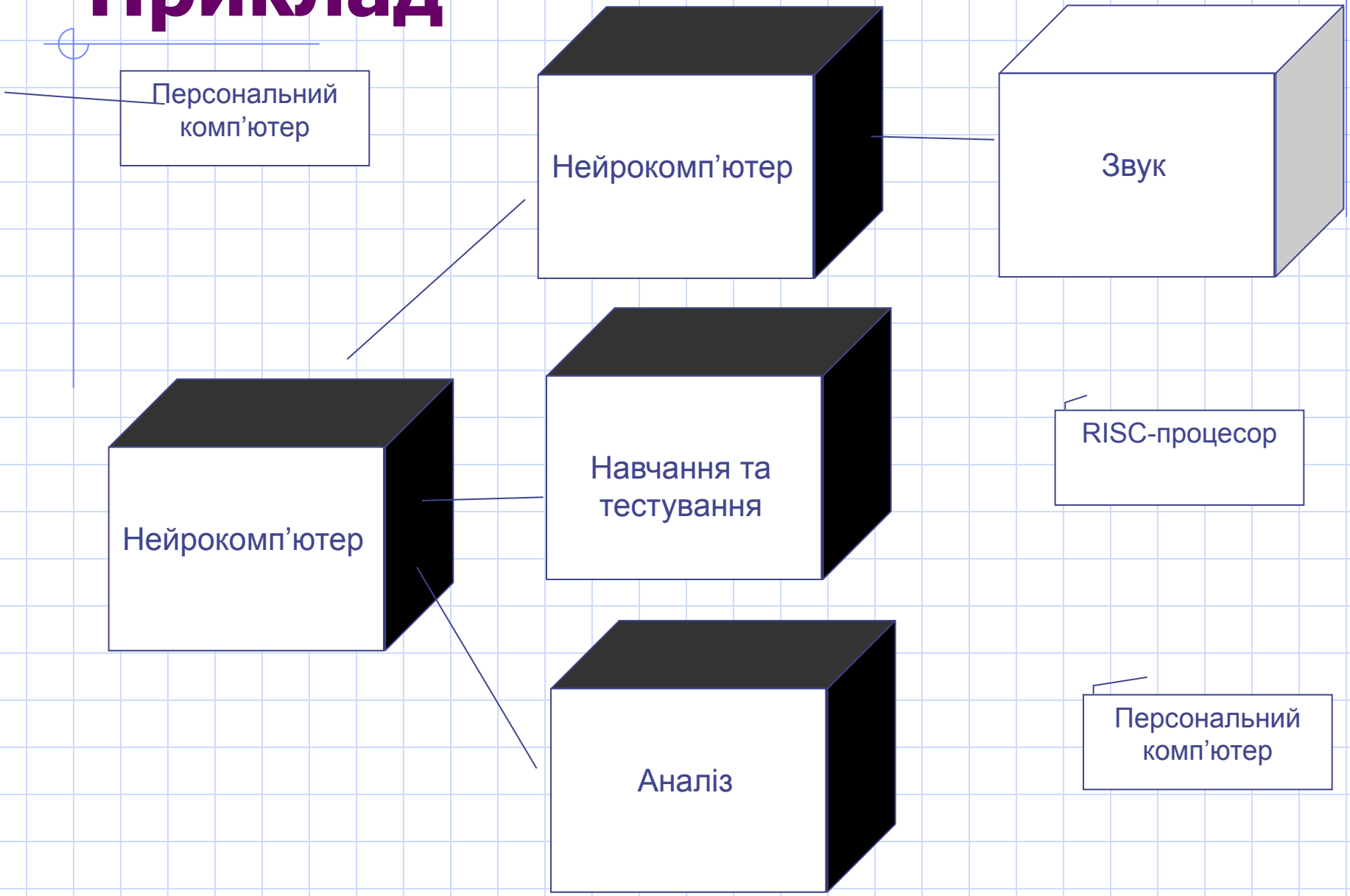
В C++ - компіляційні залежності (директива #include)

Діаграми процесів

- Відображають розподіл процесів по процесорам в физичному проекті системи
- Процесор - частина апаратури, що може виконувати програми
- Пристрій - частина апаратури, що не може виконувати програми (пристрої вводу-виводу, звуку тощо)



Приклад


Персональний комп’ютер
Нейрокомп’ютер
Звук
Нейрокомп’ютер
Навчання та тестування
RISC-процесор
Аналіз
Персональний комп’ютер

Планування процесів

- Витісняюче
  - Процес з більшим пріоритетом може віднімати процесорний час у процесу з меншим пріоритетом
- Невитісняюче
  - Однопоточність
- Циклічне
  - Процеси отримують рівну кількість процесорного часу
- Алгоритмічне
  - Використовується деякий алгоритм
- Ручне
  - Контролюється користувачем

Лекція №7

Мова UML

(Unified Modelling Language)

Основы UML

- Унифицированный язык моделирования UML=Unified Modelling Language
  - язык для определения, визуализации, конструирования и документирования артефактов программных систем, а также для моделирования экономических процессов и других не программных систем

История создания UML

- UML
  - это важный фактический и юридический стандарт для объектно-ориентированного моделирования
  - появился в 1994 году в результате совместных усилий Гради Буча и Джима Румбаха по объединению их популярных методов — метода Буча и OMT (Object Modeling Technique)
  - в 1997 году был принят в качестве стандартного языка моделирования группой промышленных стандартов OMG (Object Management Group). С тех пор он продолжает развиваться в новой редакции OMG UML.

Способы использования UML

- **Для черновиков**
  - неполные и неформальные диаграммы (зачастую нарисованные от руки на доске), создаваемые для прояснения сложных проектных решений. Здесь используется мощь визуального представления.
- **Для создания проектной документации**
  - относительно детализированные диаграммы проектирования, применяемые для визуализации и лучшего понимания существующего кода, обратного проектирования или генерации кода.
- **В качестве языка программирования**
  - полные выполняемые спецификации программных систем на языке UML. Выполняемый код можно автоматически сгенерировать. Однако разработчику его сложно осознать или модифицировать — человек работает только с "языком программирования UML". Такой способ использования UML требует отображения всей логики системы или ее поведения (возможно, с использованием диаграмм взаимодействия или состояний). Он также придает разработке робастность и надежность.

Ракурсы моделей (1)

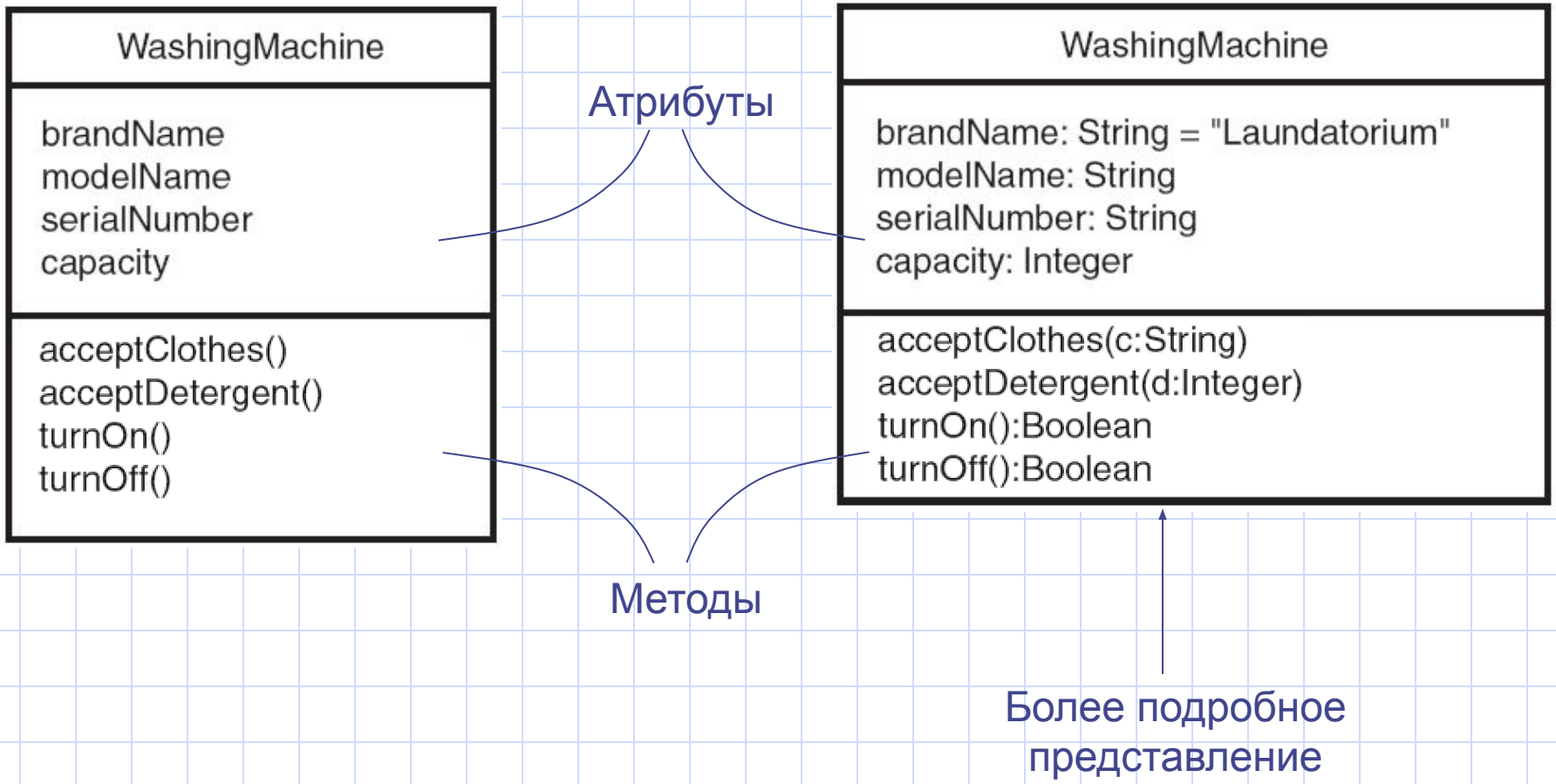
- Для всех моделей применяется одна и та же система обозначений, но интерпретация диаграмм выполняется **в разных ракурсах**:
  - *концептуальный* (conceptual perspective)
  - *ракурс спецификации* (specification perspective)
  - *ракурс реализации* (implementation perspective)

Ракурсы моделей (2)

- Концептуальный ракурс – диаграммы описывают понятия реального мира
- Ракурс спецификации – программные абстракции и компоненты со спецификациями и интерфейсами (без привязки к языку)
- Ракурс реализации – программные абстракции с привязкой к языку

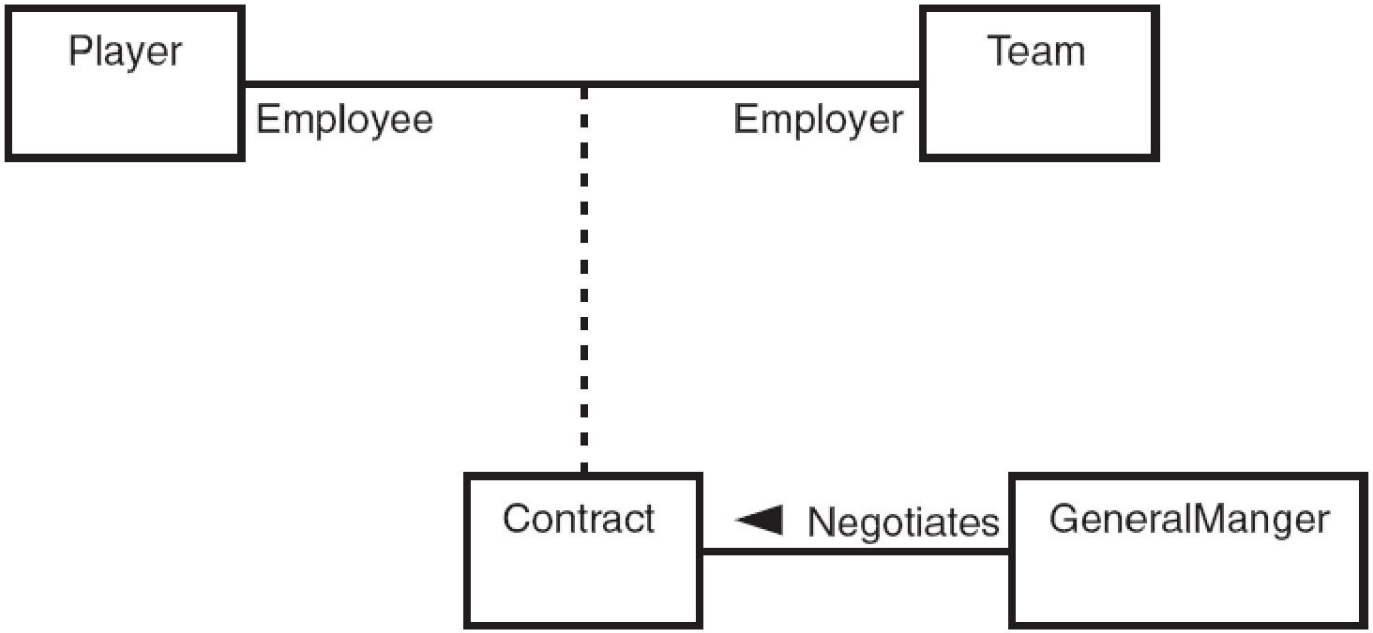
Диаграмма классов

- Показывает классы и отношения



Отношения между классами

- Ассоциация
  - Смысловая связь между классами

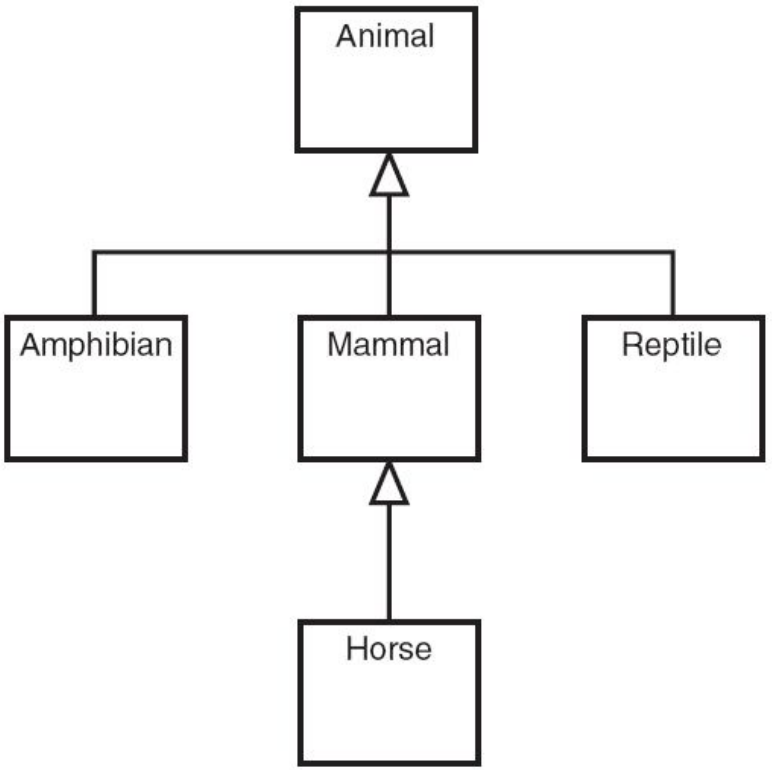


Мощность отношения

- **1** – одна связь
- **N** – неограниченное число (0 или больше)
- **0..N** - ноль или больше
- **1..N** - один или больше
- **0..1** - ноль или один
- **3..7** - заданный интервал
- **1..3, 7** - заданный интервал или точное число

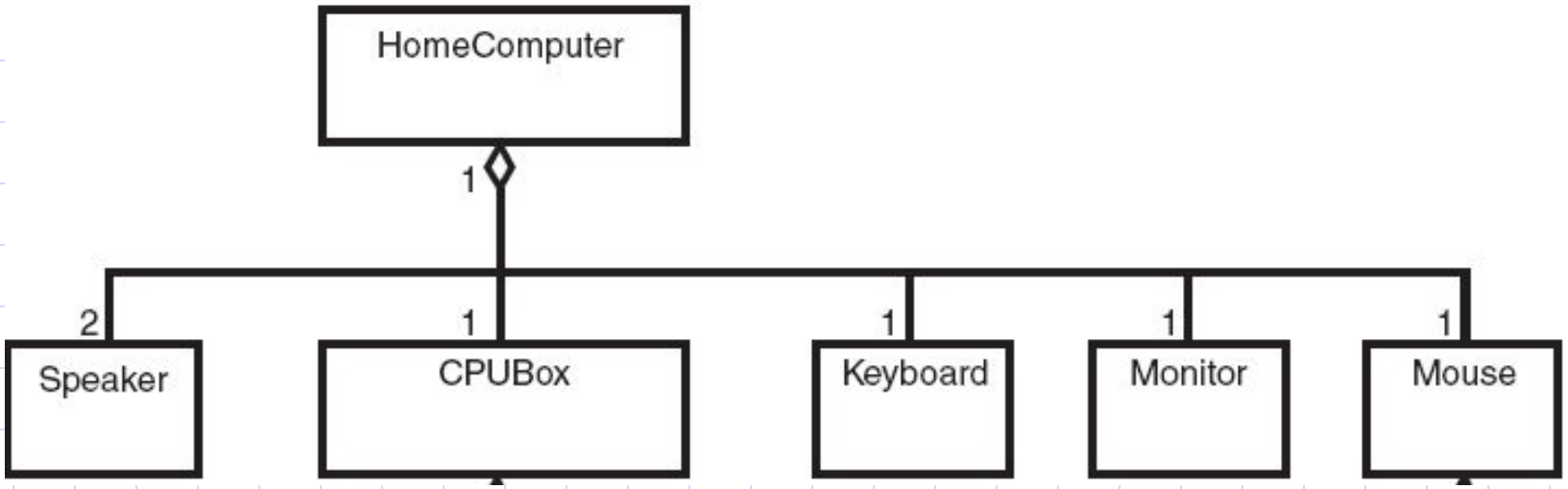
Отношения между классами

- Наследование
  - Общее-частное, обобщение



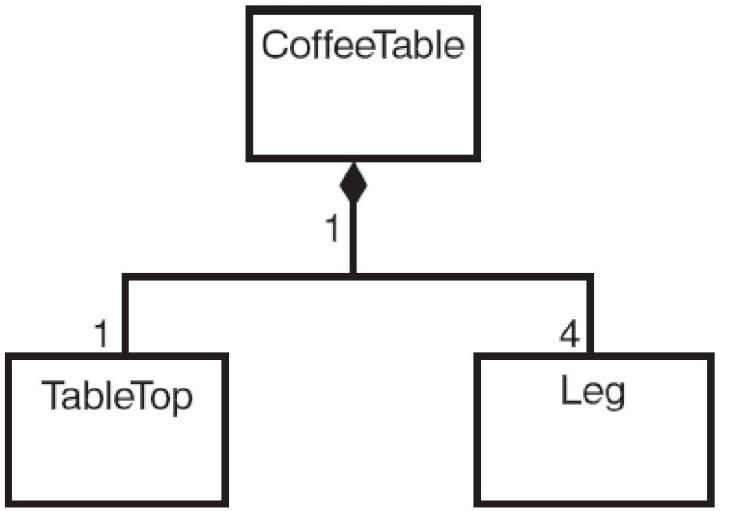
Отношения между классами

- Агрегация
  - Целое-часть



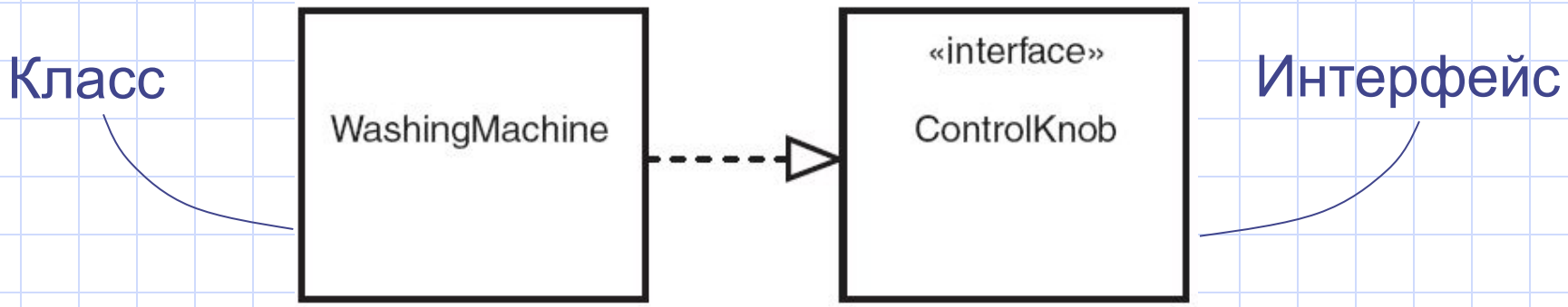
Отношения между классами

- Композит
  - строгий тип агрегации, характеризующийся тем, что каждый элемент может принадлежать только одному целому
  - Пример: компоненты кофейного столика — столешница и ножки — составляют композит



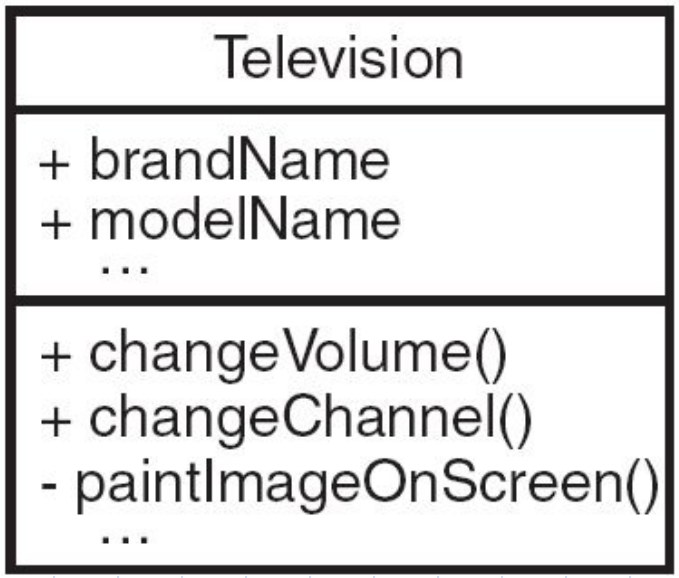
Отношения между классами

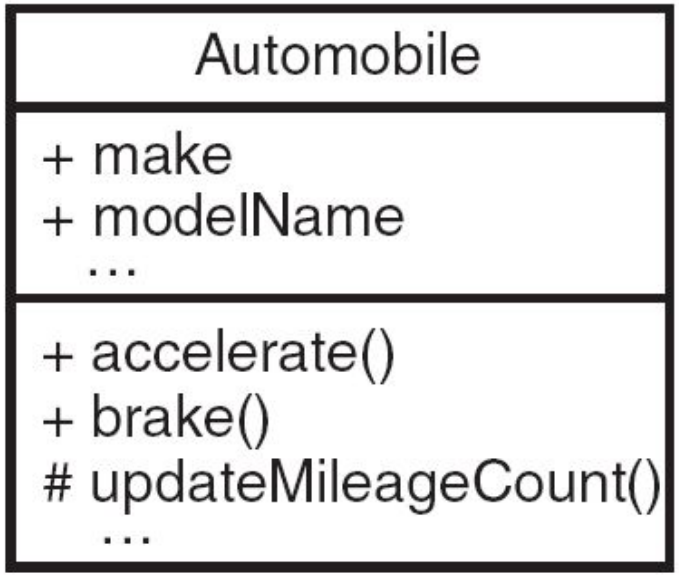
- Интерфейс
  - набор операций, которые задают некоторые аспекты поведения класса и представляют его для других классов.
- Класс связан с интерфейсом отношением *реализации*



Области видимости

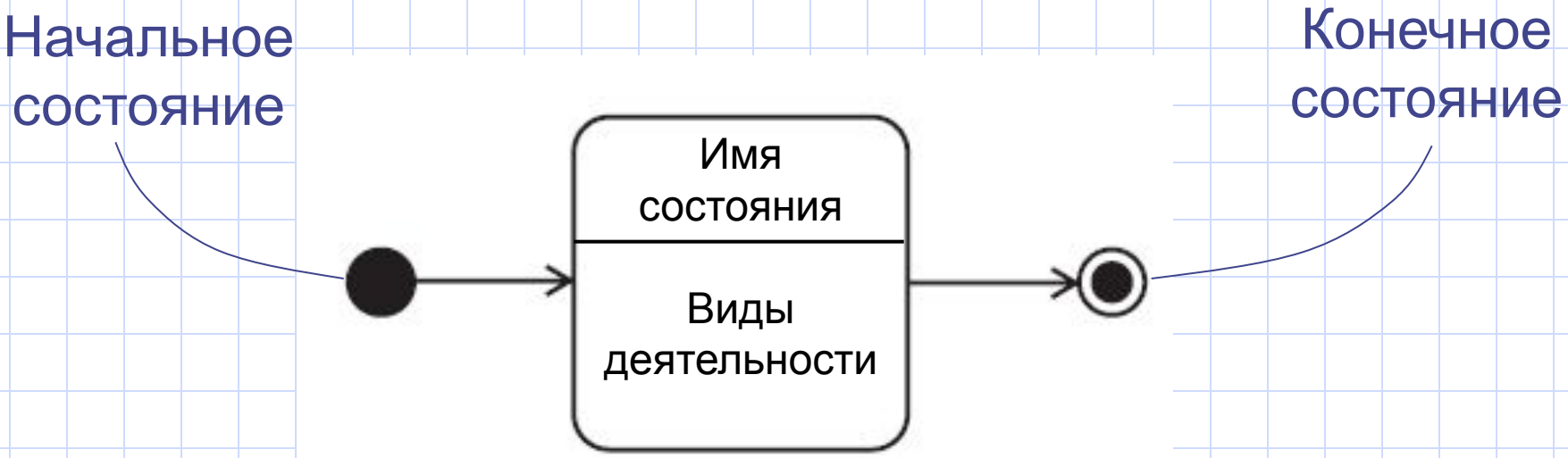
- Открытая область – public
  - “+”
- Защищенная область
  - “#”
- Закрытая область
  - “-”





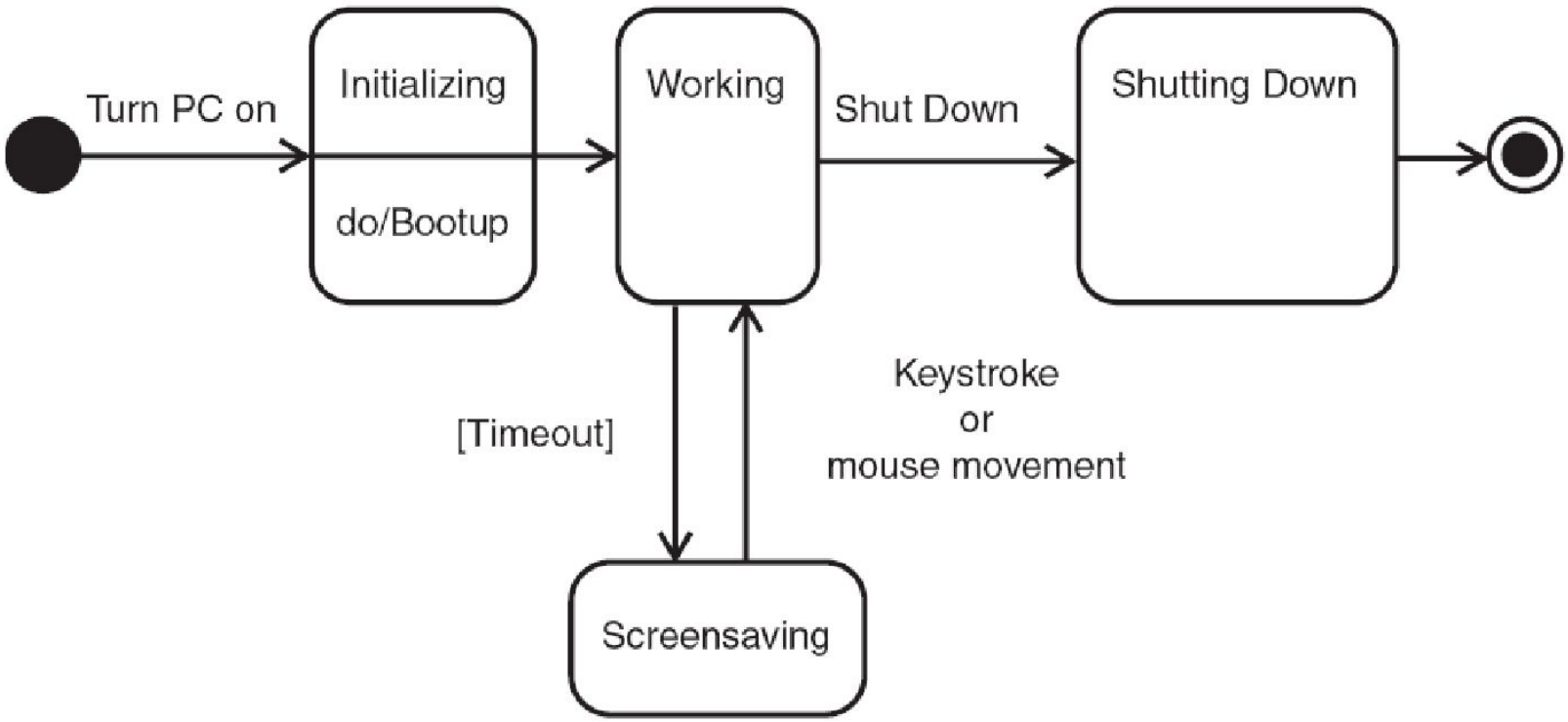
Диаграммы состояний

- Представляет состояния объекта и переходы между ними, а также показывает начальное и конечное состояние объекта



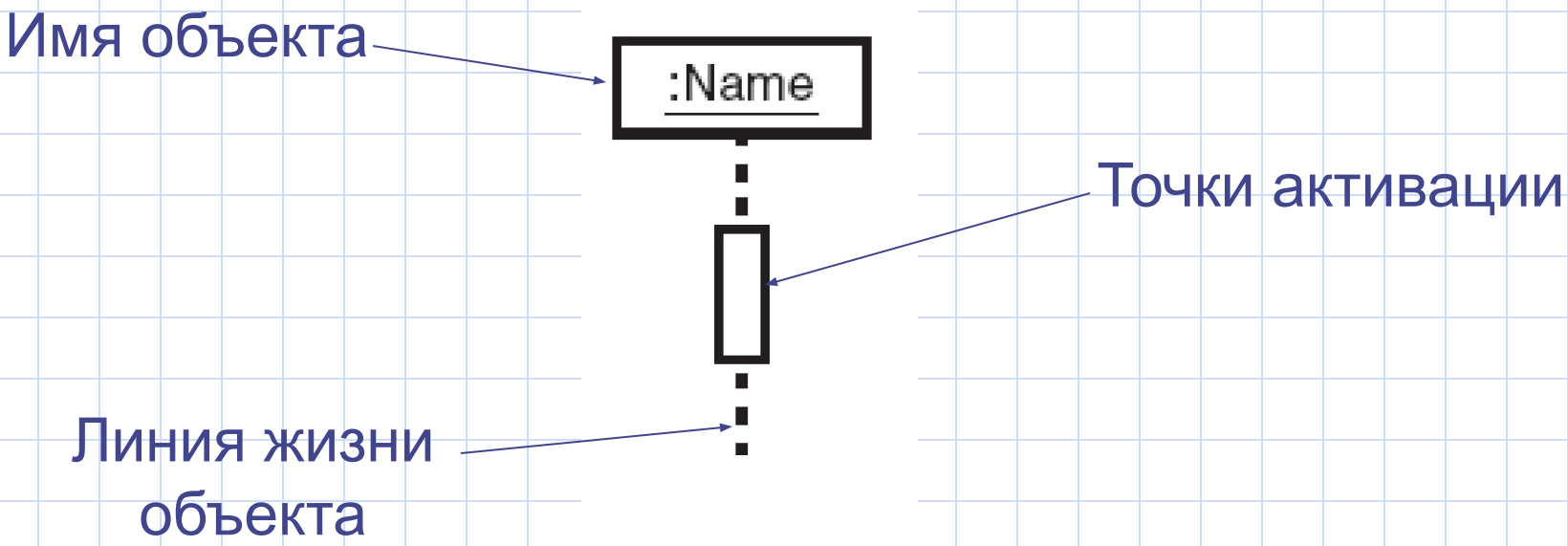
Диаграммы состояний

- Пример: класс персональный компьютер



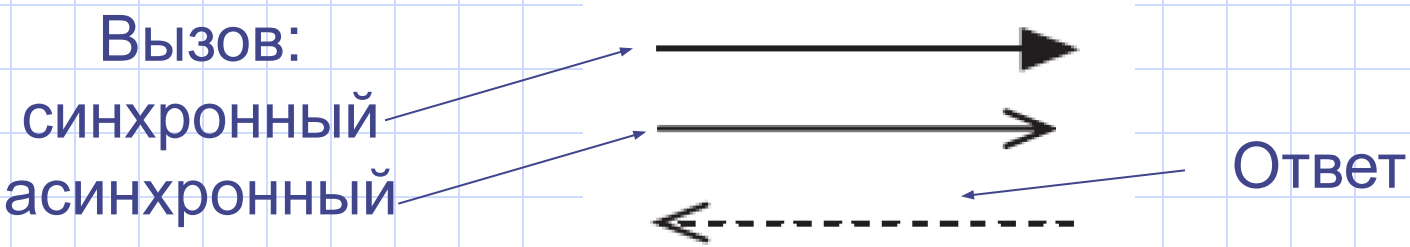
Диаграммы последовательностей

- Обеспечивает динамическое представление системы, отображая передачу сообщений между соответствующими классами



Диаграммы последовательностей

- Сообщения между объектами
  - ***вызов***: запрос объекта-отправителя к объекту-получателю на выполнение одной из его операций
    - **синхронное**: отправитель ожидает завершения выполнения операции
    - **асинхронное**: отправитель передает управление получателю и не ожидает ответа для продолжения выполнения своих действий
  - ***ответ***: ответное сообщение

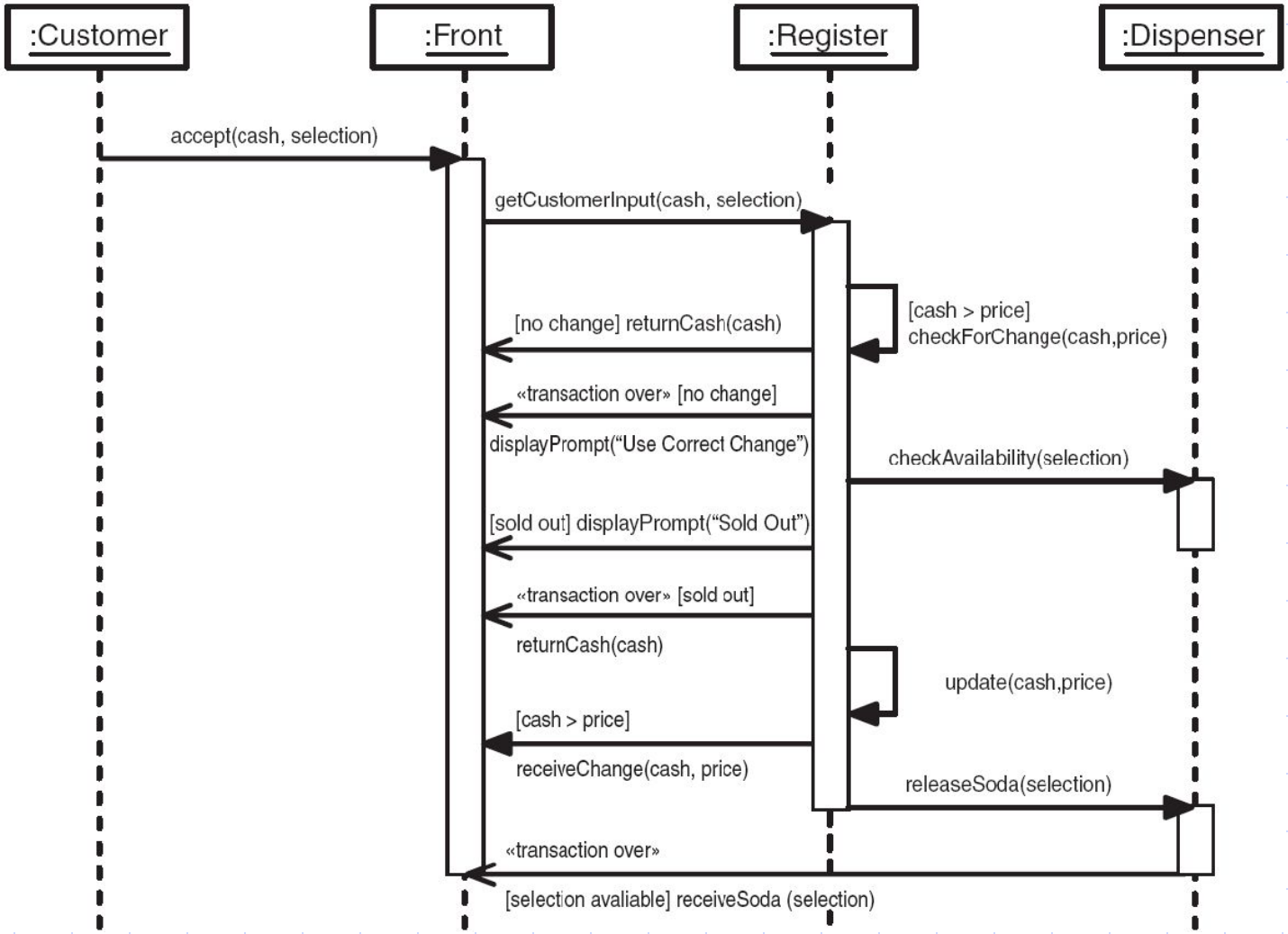


Диаграммы последовательностей

- Пример - автомат по продаже лимонада
  - 1. Покупатель помещает монету в щель на лицевой панели автомата и выбирает сорт лимонада
  - 2. Монета попадает в реестр, который ее обрабатывает.
  - 3. Реестр дает команду отсеку доставить лимонад к лицевой панели автомата.

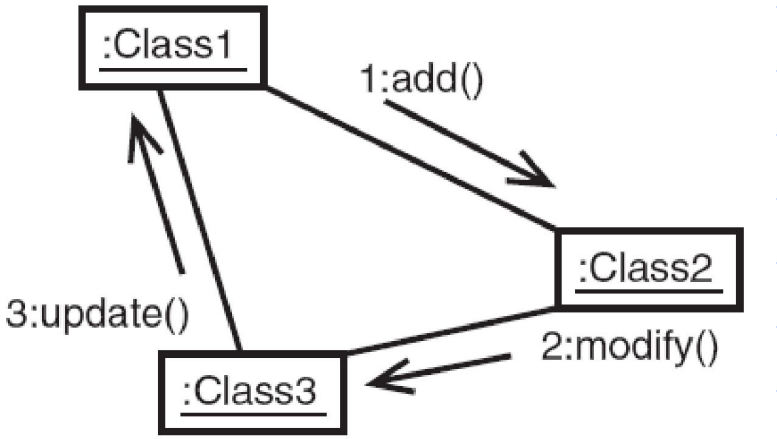
Диаграммы последовательностей

- Пример - автомат по продаже лимонада



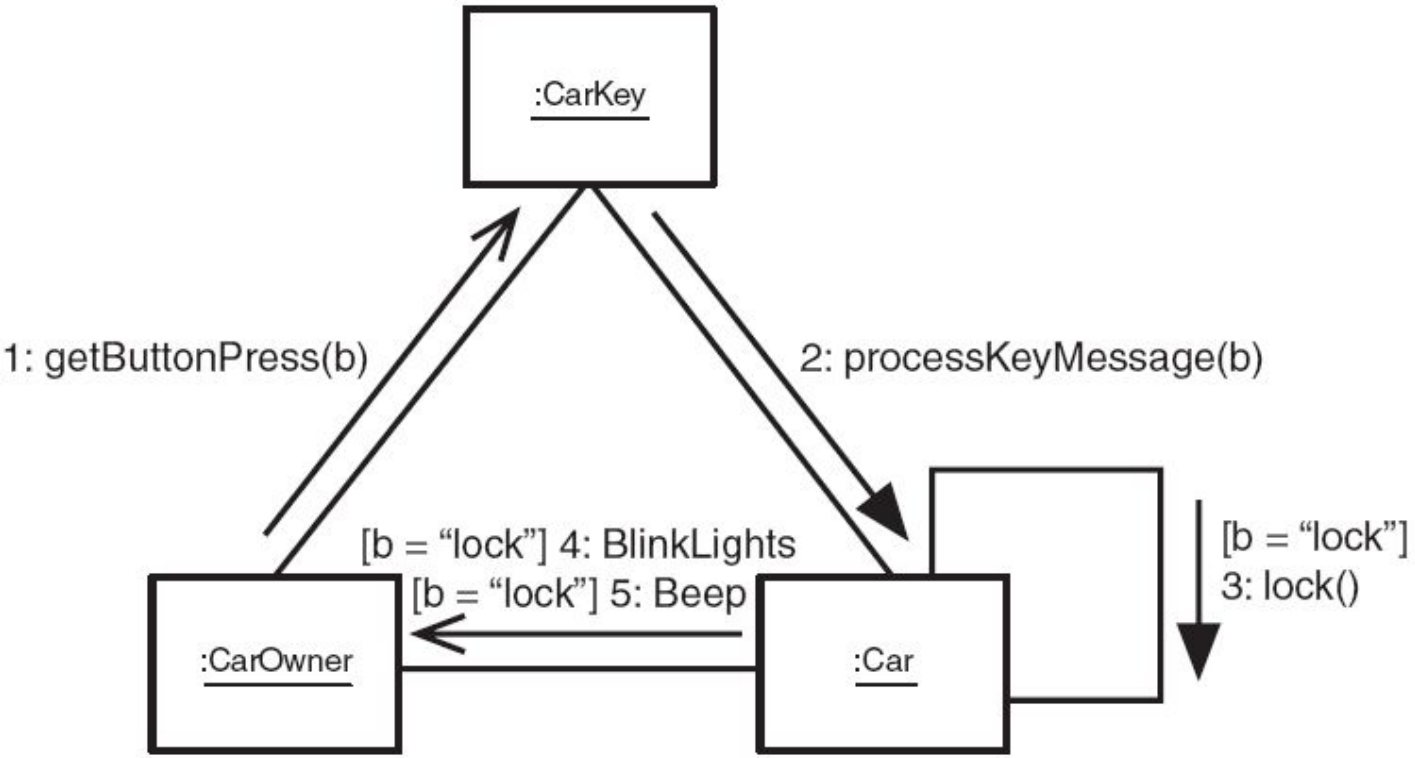
Диаграммы коммуникации

- Являются расширением понятия объектной диаграммы - в дополнение к связям между объектами включают сообщения, которые объекты передают друг другу
- Отражает взаимодействие объектов во времени
- Аналог диаграммы объектов в нотациях Буча



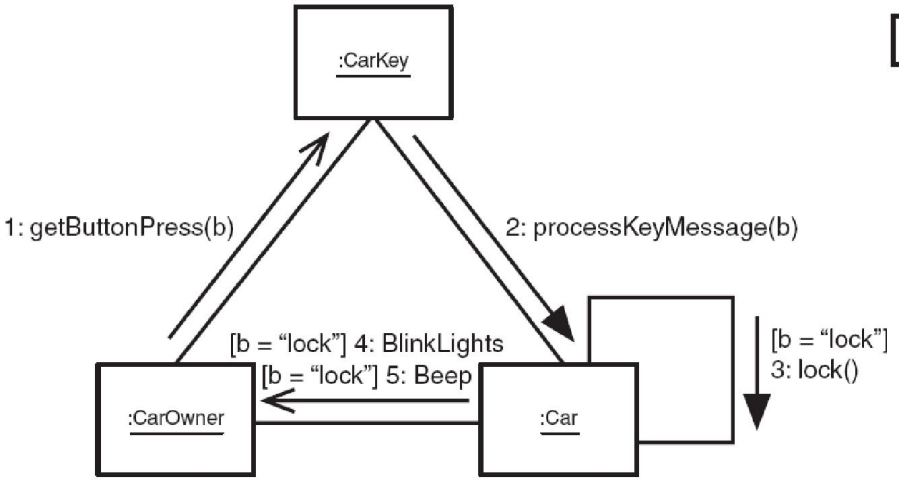
Диаграммы коммуникации

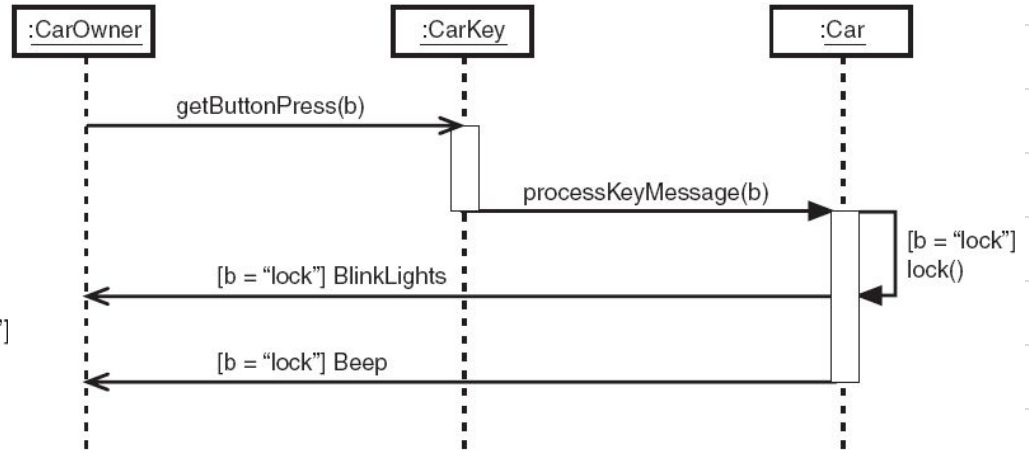
- Пример 1 - автомобили и ключи



Диаграммы коммуникации и последовательностей

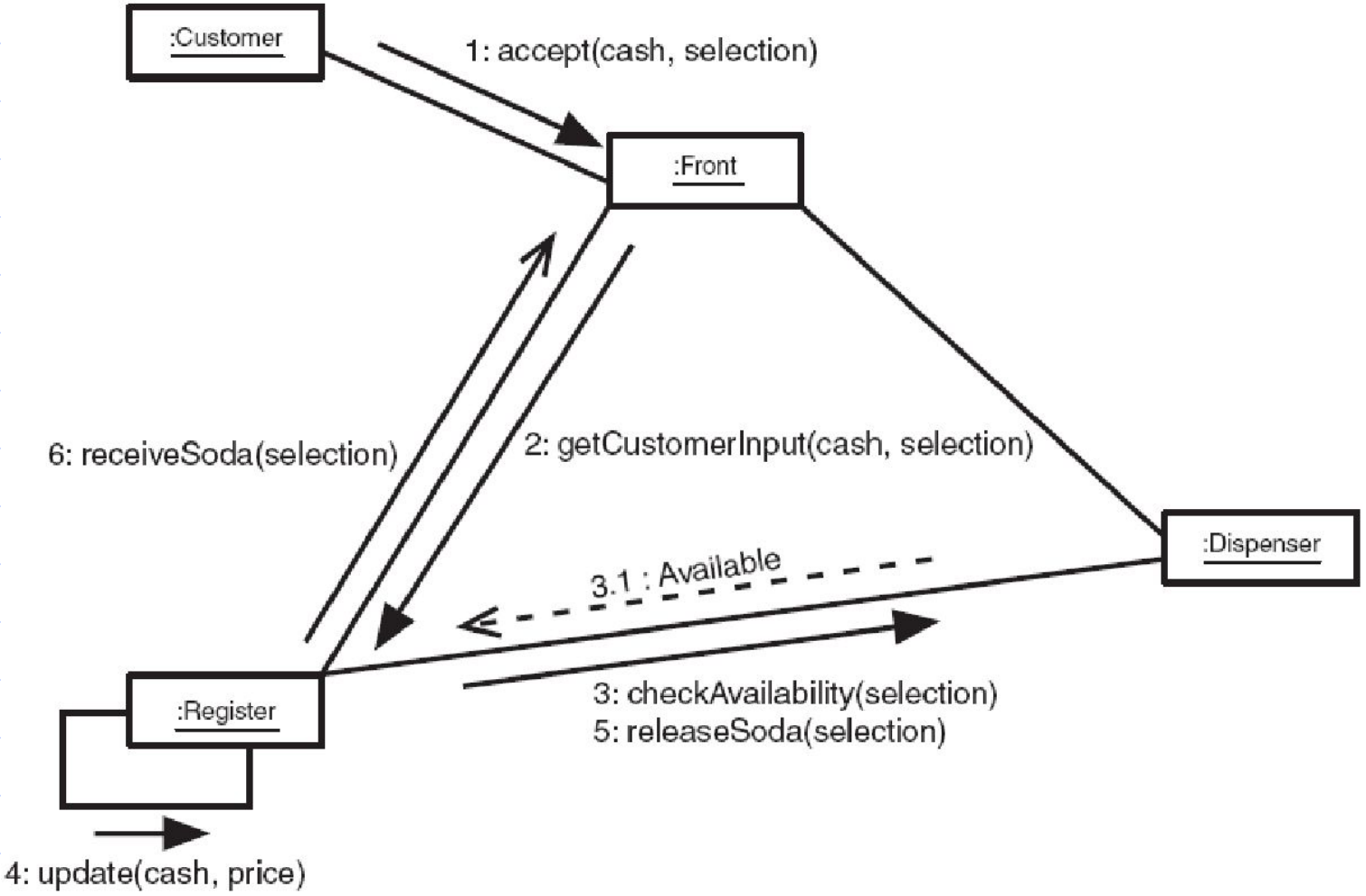
- Любую диаграмму последовательностей можно преобразовать в диаграмму коммуникации, и наоборот





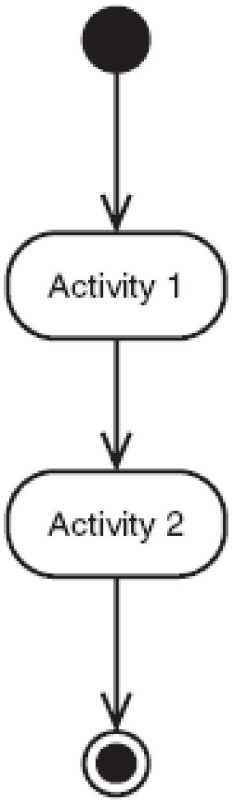
Диаграммы коммуникации

- Пример 2 - автомат по продаже лимонада

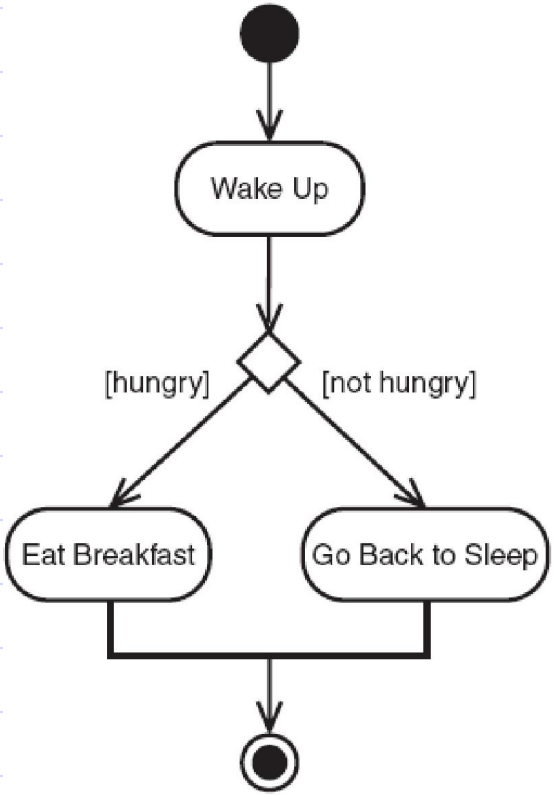


Диаграммы видов деятельности

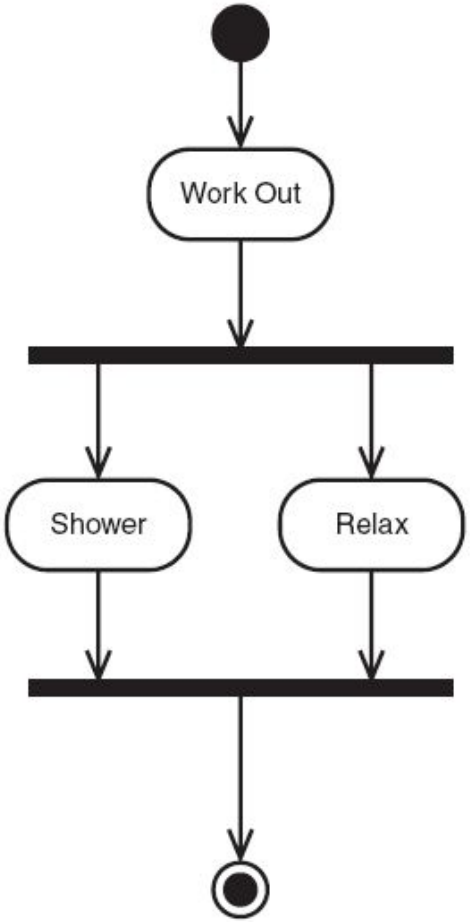
- Описывается последовательность шагов (*видами деятельности*)
- Похожа на старые блок-схемы
- Удобна для отображения бизнес-процессов или операций



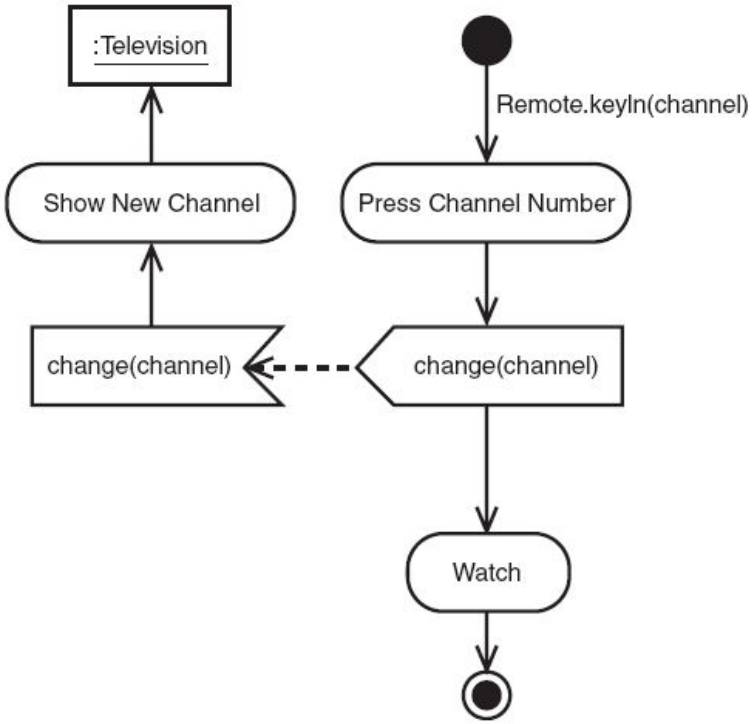
Диаграммы видов деятельности



Принятие решения



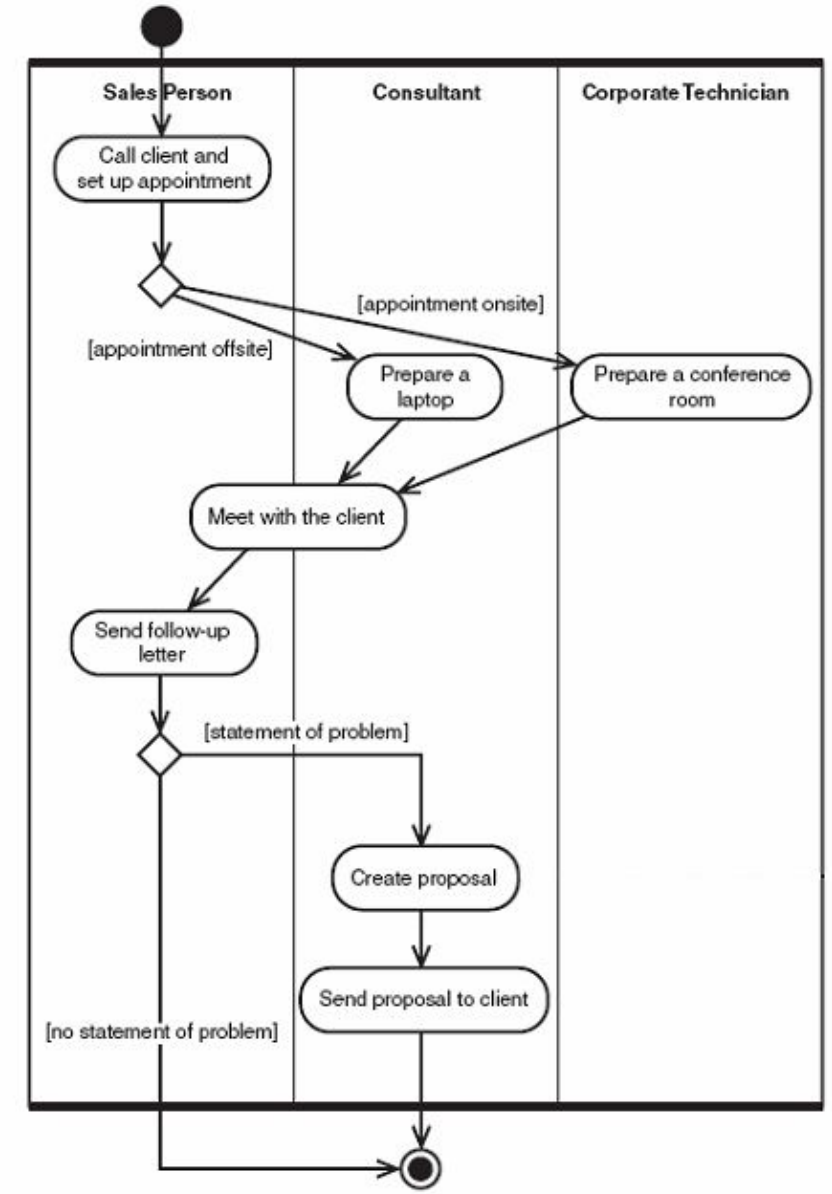
Параллельные пути развития событий



Отправка и получение сигналов

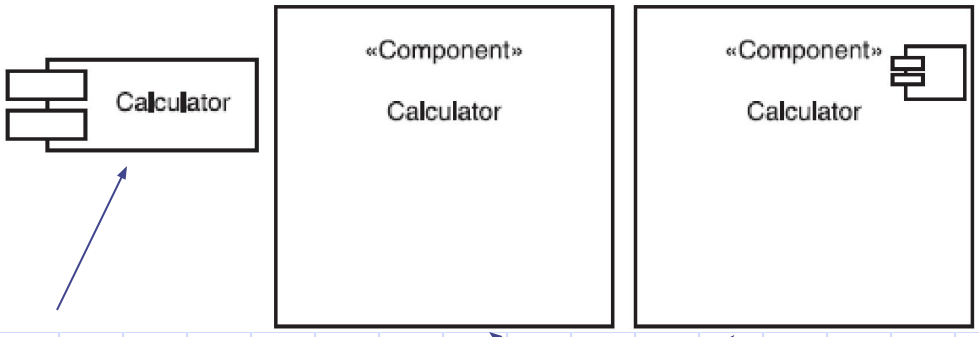
Диаграммы видов деятельности

- Пример - диаграмма видов деятельности для бизнес-процесса встречи с новым клиентом
  - 1.Агент договаривается с клиентом о встрече
  - 2. Если встреча планируется в офисе консалтинговой фирмы, то специалист по техническому обеспечению готовит конференц-зал для переговоров
  - 3. Если встреча состоится в офисе клиента, консультант готовит всю необходимую документацию на переносном компьютере
  - 4. Консультант и агент встречаются с клиентом в назначенном месте и в оговоренное время
  - 5. Агент готовит документы о результатах встречи
  - 6. Если результат встречи положителен и задача сформулирована, консультант оформляет предложение и отправляет его клиенту



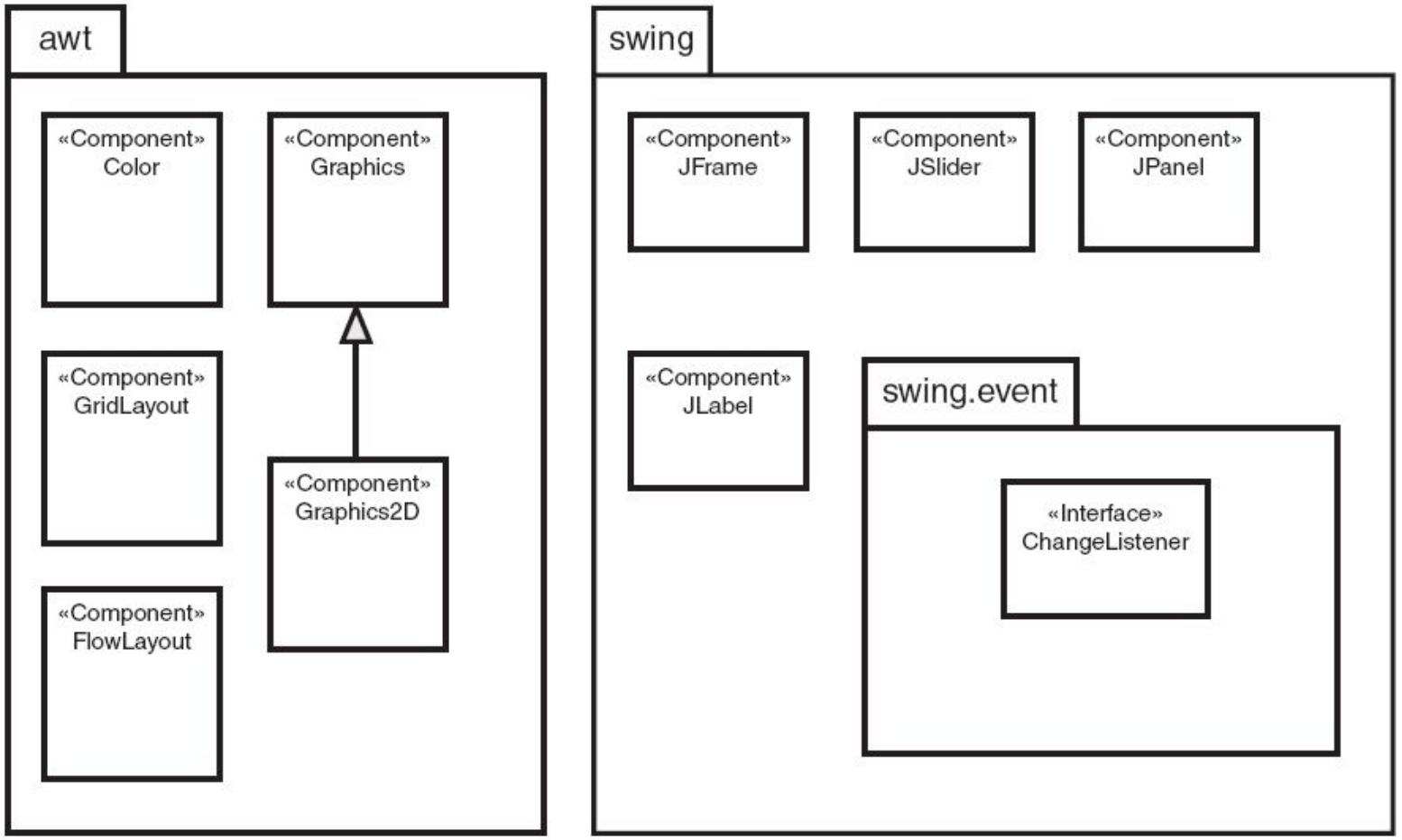
Диаграммы компонентов

- Компонент
  - является модулем или частью системы
  - определяет функциональность системы: представляет собой программную реализацию одного или нескольких классов



Диаграммы компонентов

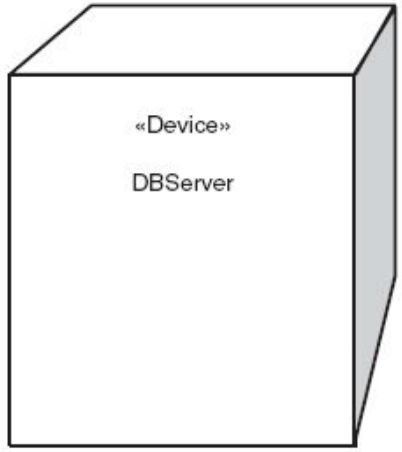
- Пример: библиотеки Java для работы с графикой



Диаграммы развертывания

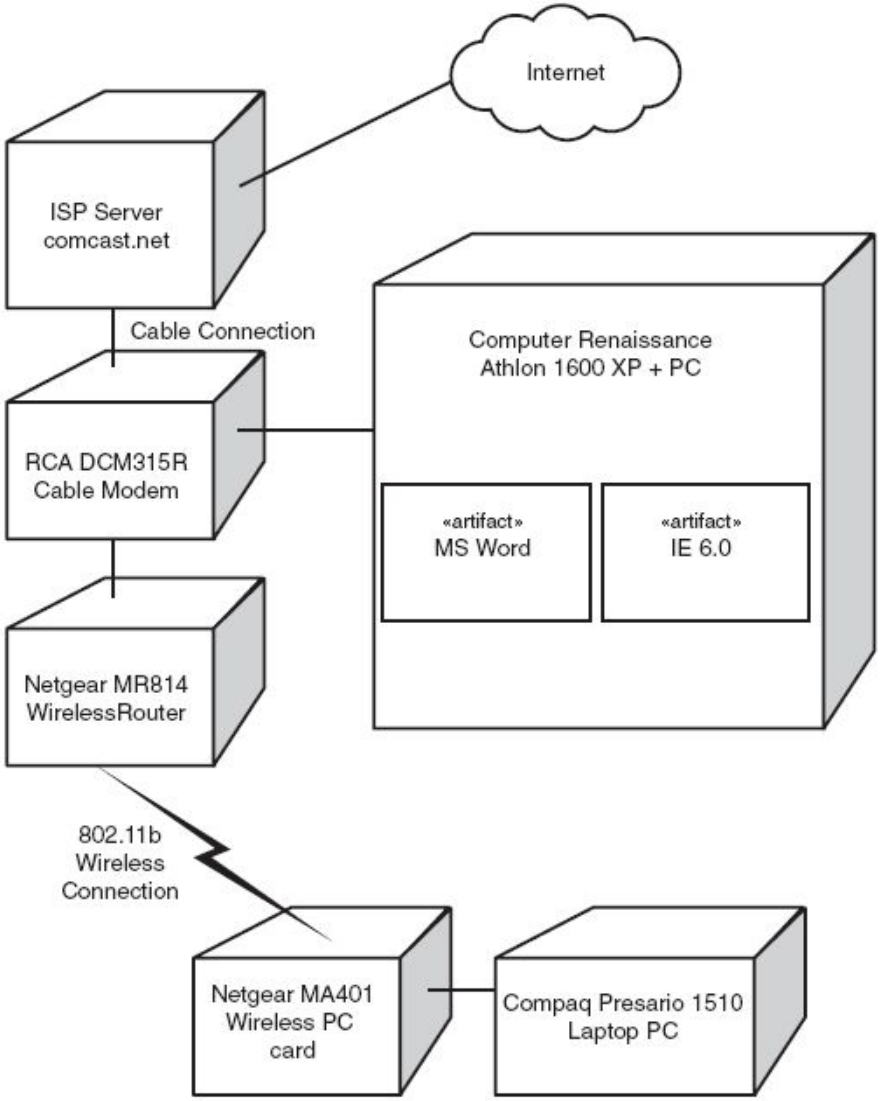
- Отражают способ воплощения артефактов в физической системе и способ соединения аппаратных средств между собой
- Главным аппаратным элементом является *узел*
- Аналог д. процессов в нотациях Буча

Представление узла в UML



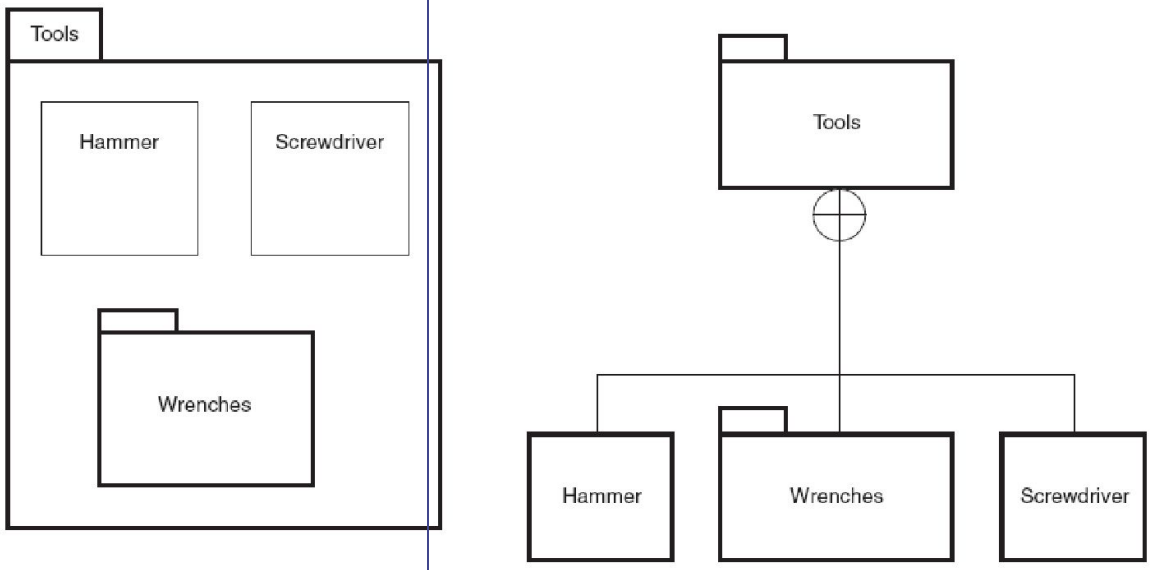
Диаграммы развертывания

- Диаграмма развертывания для домашней системы



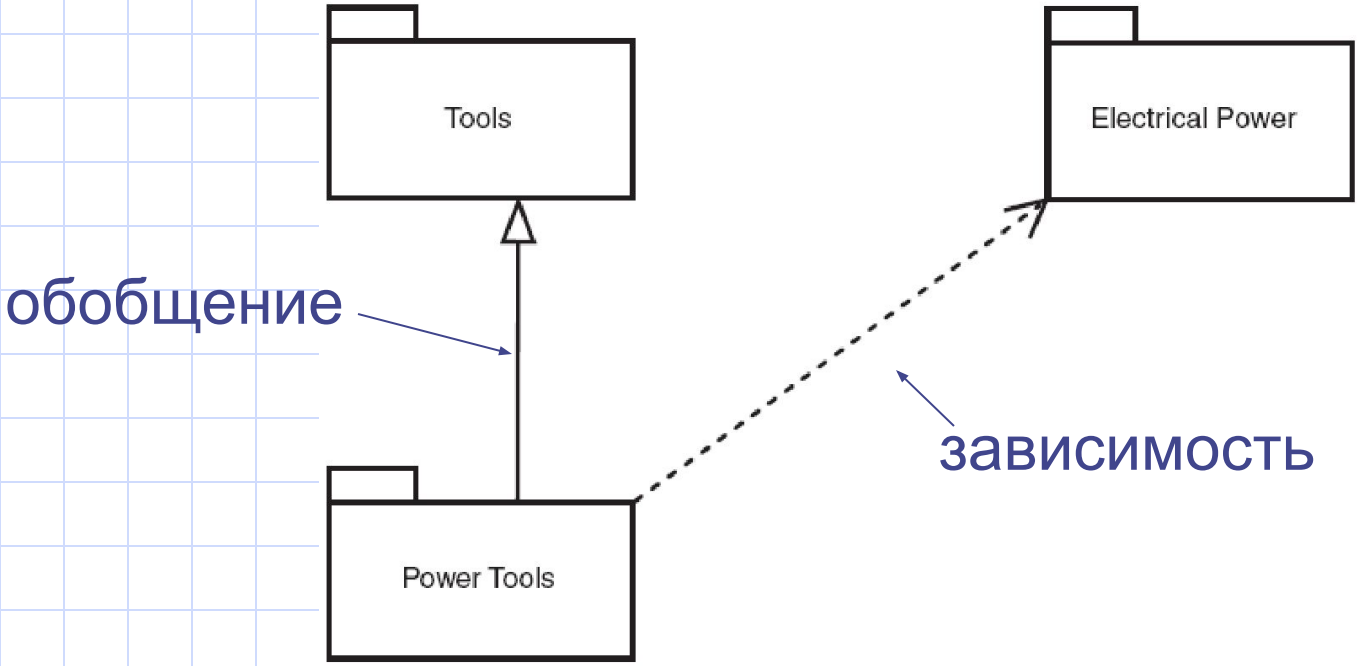
Диаграммы пакетов

- Пакеты предназначены для группирования элементов диаграмм (например, классов)
- Если пакету присвоено имя, значит, это же имя связано и с группой элементов
- В терминах языка UML пакет предоставляет для *содержащихся* в нем элементов *пространство имен*



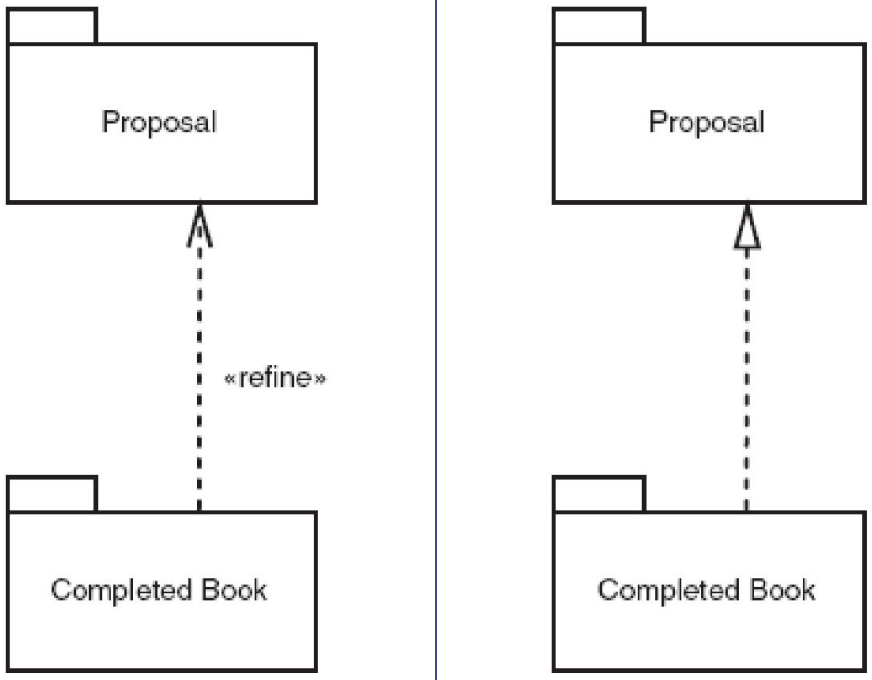
Отношения между пакетами

- *обобщение*,
- *зависимость* или
- *уточнение*
  - один пакет уточняет другой, если в нем содержатся те же самые элементы, но в более подробном представлении



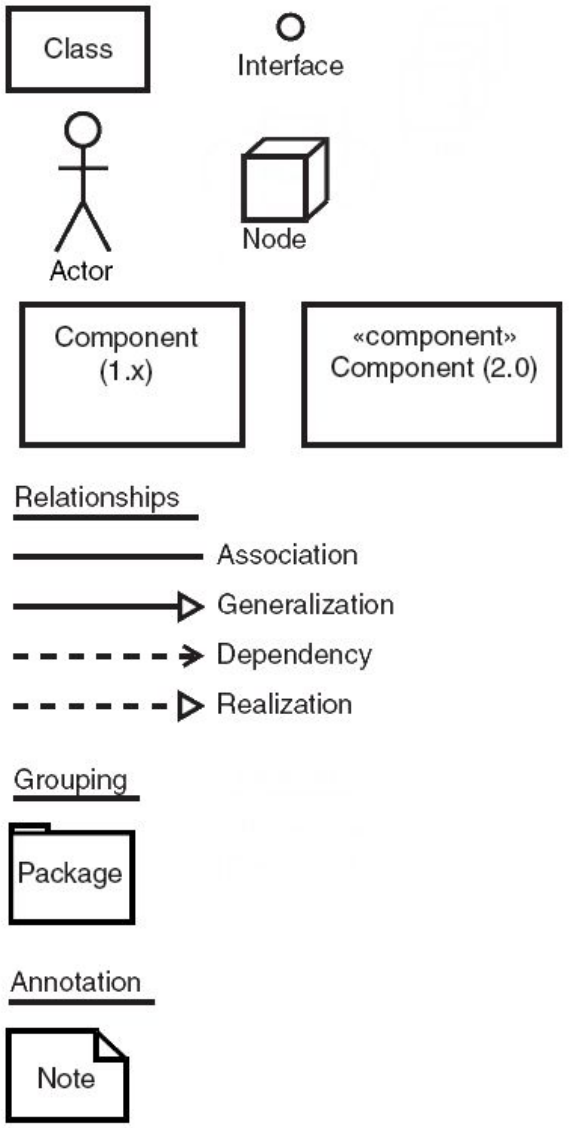
Отношения между пакетами

- Два способа визуализации отношения уточнения



Резюме

Структурные элементы



Динамические элементы