Классы и объекты

Александра Волосова,
к.т.н., доцент кафедры
ИУ5

Презентация разработана в рамках гранта «Грант на обучение студентов по образовательным программам высшего образования для топ-специалистов в сфере информационных технологий. Договор № 70-2025-000850 с АНО «Аналитический центр при Правительстве Российской Федерации»

Часть 1

Объект. Класс

Объект

С точки зрения восприятия человеком объектом может быть:

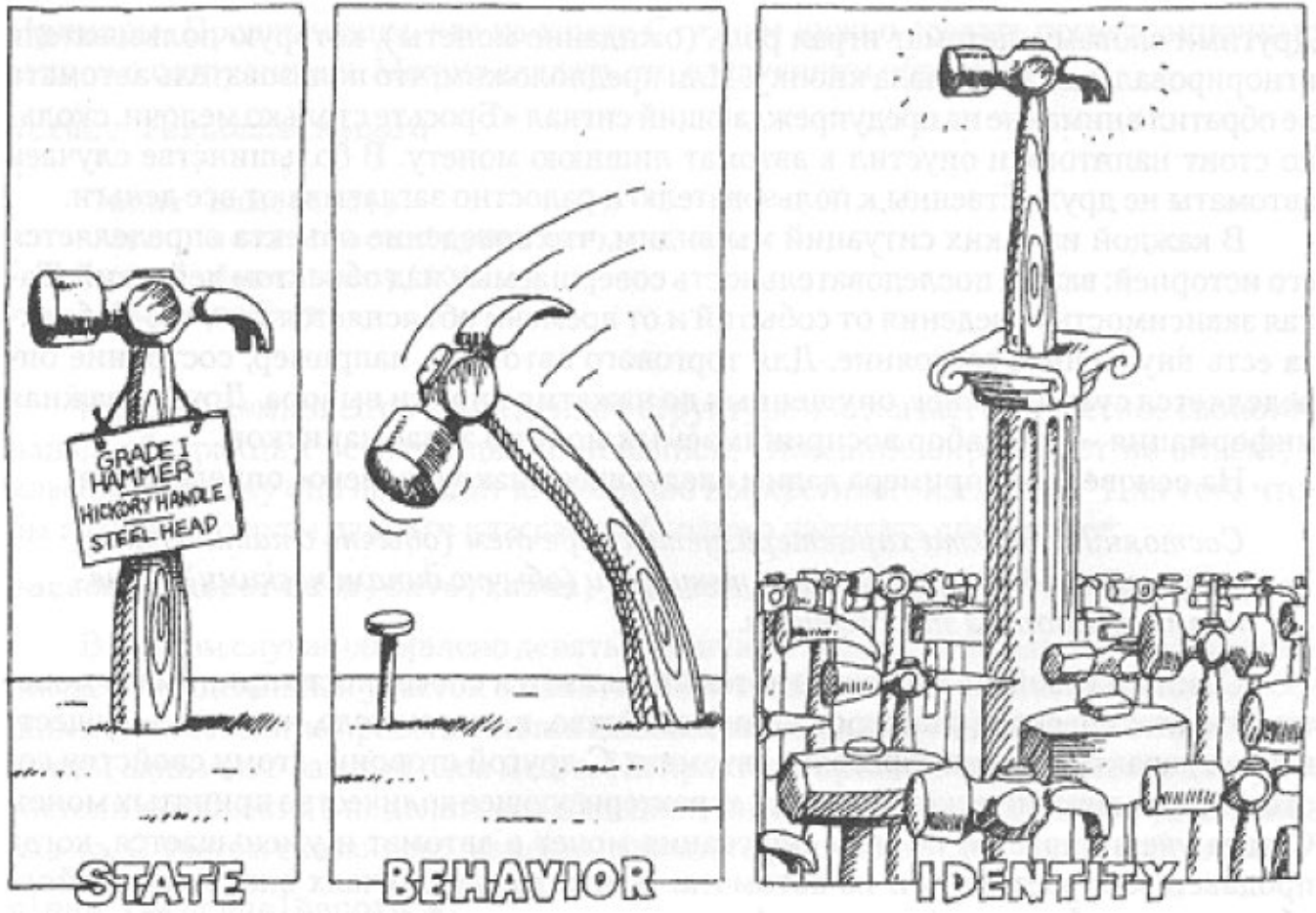
- осязаемый и (или) видимый предмет;
- нечто, воспринимаемое мышлением;
- нечто, на что направлена мысль или действие.

Термин **объект** в программном обеспечении впервые был введен в языке Simula и применялся для моделирования реальности

Объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области

3.1. Природа объекта

- Объект обладает состоянием, поведением и идентичностью
- Состояние: свойства и их текущие значения
- Поведение: как объект действует и реагирует
- Идентичность: что отличает объект от других
- Структура и поведение определяются классом



Объект обладает состоянием, поведением и идентичностью
Структура и поведение схожих объектов определяет общий для них класс

Состояние объекта (State)

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

Состояние объекта

- Характеризуется свойствами объекта
- Свойства могут быть статическими или динамическими
- Пример: торговый автомат — сумма денег, запас напитков
- Пример: лифт — текущий этаж, направление движения
- Состояние может включать ссылки на другие объекты

```
struct PersonnelRecord {
char name[100];
int socialSecurityNurober;
char department[10];
float salary;
};
PersonnelRecord   deb, karen
```

```
class PersonnelRecord {
public:
char* employeeName() const;
int employeeSocialSecurityNumberf) const;
char* employeeDepartment() const;
protected:
char name[100];
int socialSecurityNumber;
char department[10];
float salary; };
```

Поведение (Behavior)

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

Понятие *сообщение* совпадает с понятием *операции* над объектами. В ООпрограммировании эти два термина могут использоваться как синонимы

Поведение объекта

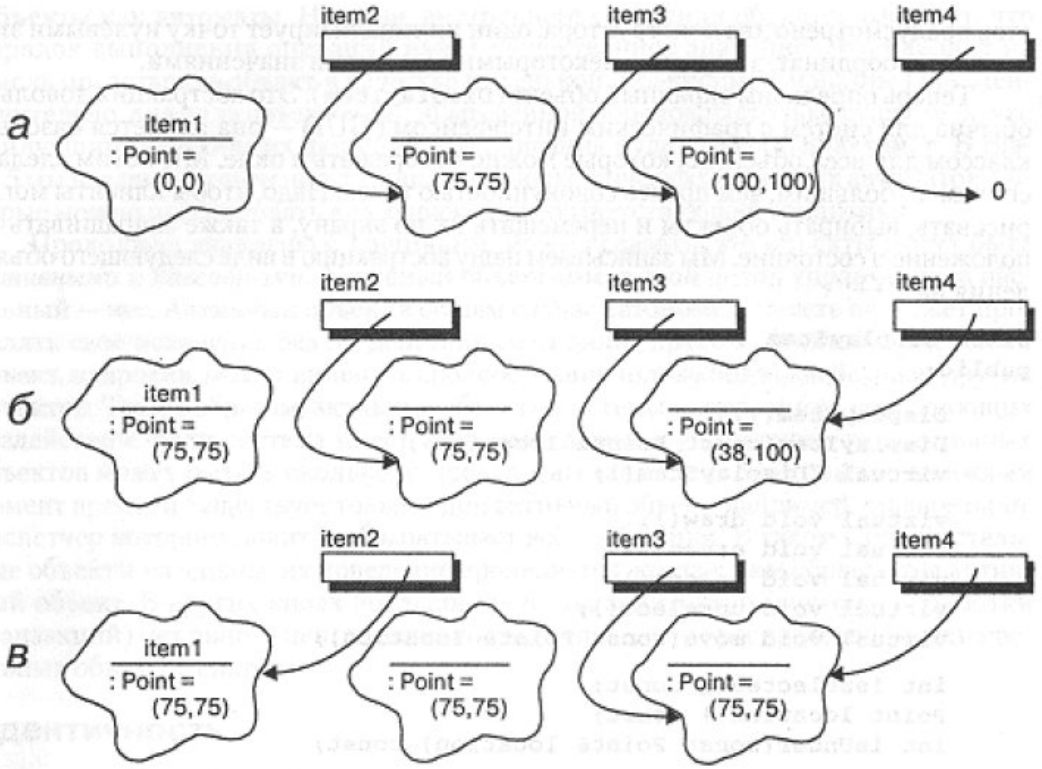
- Определяется операциями над объектом
- Зависит от состояния объекта
- Пример: автомат принимает монеты только в состоянии "ожидание"
- Операции = методы = функции-члены = сообщения

Примеры: Конструктор - операция создания объекта и/или его инициализации.
Деструктор - операция, освобождающая состояние объекта и/или разрушающая сам объект.

- Внешнее проявление внутренней логики

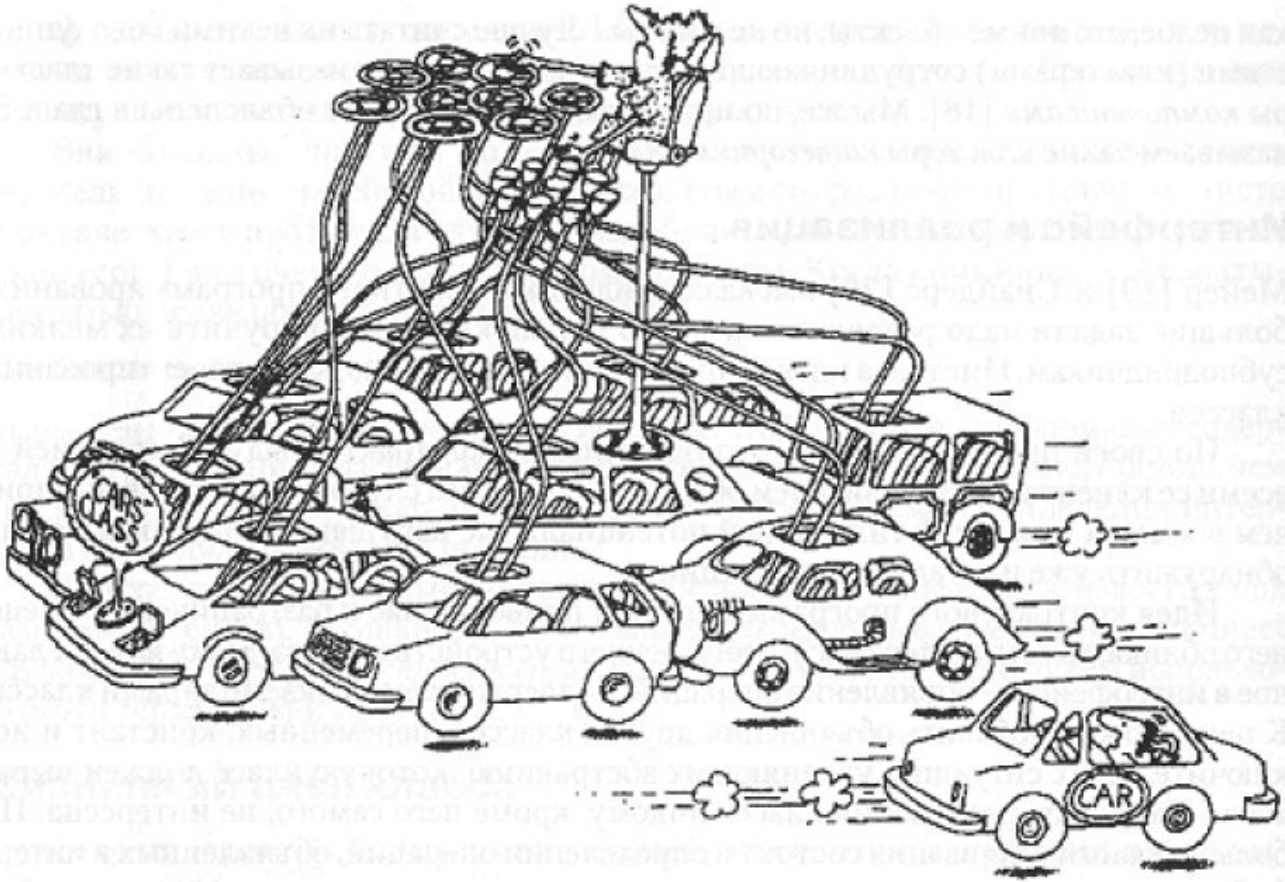
Идентичность (Identity)

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




```
DisplayItem     item1;
DisplayItem*    item2 = new DisplayItem(Point(75, 75));
DisplayItem*    item3 = new DisplayItem(Point(100, 100));
DisplayItem*    item4 = 0;
```

Класс



Класс - набор объектов, которые обладают общей структурой и одинаковым поведением

Пример класса PersonnelRecord

```
class PersonnelRecord {
  public:
    char* employeeName() const;
    int employeeSocialSecurityNumber() const;
    char* employeeDepartment() const;
  protected:
    char name[100];
    int socialSecurityNumber;
    char department[10];
    float salary;
};
```

- Инкапсуляция данных
- Разделение интерфейса и реализации
- Контроль доступа: public, protected

Природа классов

Класс — это множество объектов с общими свойствами

- Определяет структуру (данные) и поведение (операции)
- Абстракция, обобщающая схожие объекты
- Интерфейс + реализация + общие для всех экземпляров элементы

Метафоры класса

- Шаблон или форма для создания объектов
- Штамп для печати экземпляров
- Категория или множество объектов
- Описание общих характеристик
- В программировании: тип данных + операции

Интерфейс класса

Можно разделить интерфейс класса на три части:

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

```
class имя_класса
{
// компоненты класса
};
```

```
class Person { };

int main()
{

}
```

```
class Person
{

};
int main()
{
    Person person;
}
```

```
#include <iostream>
#include <string>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;
    // устанавливаем значения полей класса
    person.name = "Tom";
    person.age = 38;
    // вызываем функцию класса
    person.print();
}
```

объект.компонент

```
person.name = "Tom";
person.age = 38;
person.print();
```

```
#include <iostream>
#include <string>
class Person
{
public:
   std::string name;
   unsigned age;
   void print()
   {
      std::cout << "Name: " << name << "\tAge: " << age <<
std::endl;
   }
};
int main()
{   Person person;
   // устанавливаем значения полей класса
   person.name = "Bob";
   person.age = 42;
   // получаем значения полей
   std::string username = person.name;
   unsigned userage = person.age;
   // выводим полученные данные на консоль
   std::cout << "Name: " << username << "\tAge: " << userage
<< std::endl;
}
```

Пример класса

```
class PersonnelRecord {
public:
char* employeeName() const;
int employeeSocialSecurityNumberf) const;
char* employeeDepartment() const;
protected:
char name[100];
int socialSecurityNumber;
char department[10];
float salary;
};
```

В классе:

- Обеспечивается контроль доступа
- Реализуется инкапсуляция
- Разделяется интерфейс и реализация

Часть 2

Отношения между объектами

Отношения между классами

Отношения между объектами

два типа иерархических отношений между объектами:

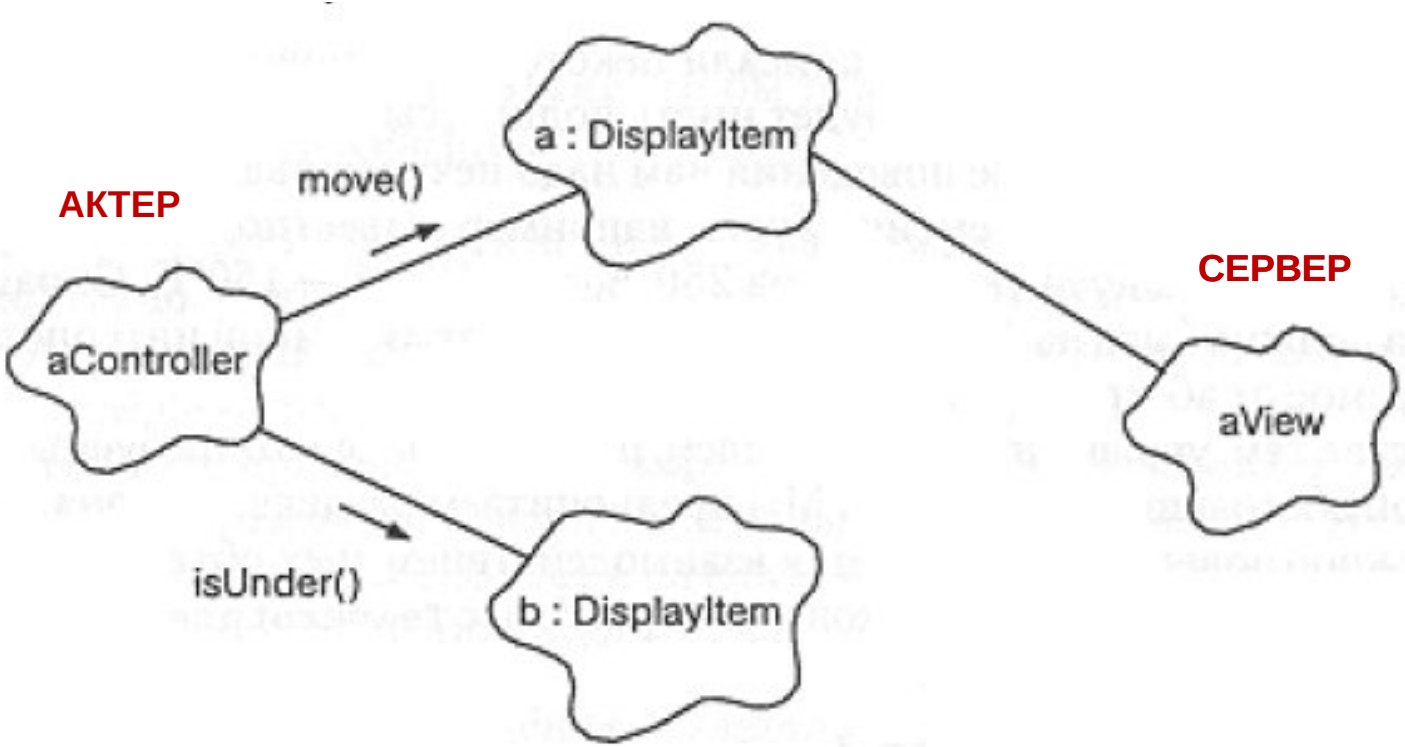
- связи
- агрегация

Связи : физические или концептуальные соединения

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

- **Актер**: Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов
- **Сервер**: Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта
- **Агент:** Такой объект может выступать как в активной, так и в пассивной роли. Объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или агента.

Связи


АГЕНТ
a : DisplayItem
move()
АКТЕР
СЕРВЕР
aController
aView
isUnder()
b : DisplayItem

Пример. В промышленных процессах требуется непрерывное изменение температуры. Необходимо поднять температуру до заданного значения, выдержать заданное время и понизить до нормы. Профиль изменения температуры у разных процессов разный; зеркало телескопа надо охлаждать очень медленно, а закаляемую сталь очень быстро.

Абстракция нагрева имеет достаточно четкое поведение. Поэтому можно описать класс:

// тип, значение которого задает прошедшее время в минутах

typedef unsigned int Minute; //

Класс **TemperatureRamp** задает функцию времени от температуры:

```
class TemperatureRamp {
public:
TemperatureRamp() ;
virtual ~TenperatureRamp() ;
virtual void clear();
virtual void bind (Temperature, Minute);
Temperature TemperatureAt(Minute);
protected:
...
};
```

Нужное поведение достигается взаимодействием трех объектов:

- экземпляра **TemperatureRamp**,
- нагревателя и контроллера

Класс **TemperatureController** :

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

Операция **process** обеспечивает основное поведение этой абстракции; ее назначение - передать график изменения температуры нагревателю, установленному в конкретном месте.

TemperatureRamp growingRamp;

TemperatureController rampController(7) ;

Создание точек и загрузка плана в контроллер:

growingRamp. bind (250, 60);

growingRamp.bind(150, 180);

rampController.process(growingRamp) ;

rampController - агент, ответственный за выполнение температурного плана. Агент использует объект **growingRamp,** как сервер. Проявление связи: **rampController** явно получает **growingRamp** в качестве параметра одной из своих операций.

// Число, однозначно определяющее положение датчика typedef unsigned int Location;

Объект **rampController** видит объект **growingRamp**, так как они находятся в одной области видимости и потому, что **growingRamp** передается объекту **rampController** в качестве параметра.

Существуют следующие способы обеспечения видимости:

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

Выбор способа зависит от тактики проектирования.

Виды отношений между объектами

1. Использование (uses): клиент-сервер
2. Создание (creates): фабричные методы
3. Наследование (inherits): is-a отношение
4. Агрегация (has-a): part-of отношение

• Степень отношений: один-к-одному, один-ко-многим

Отношения между объектами

два типа иерархических отношений между объектами:

- связи
- агрегация

Агрегация : агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно.

Пример:

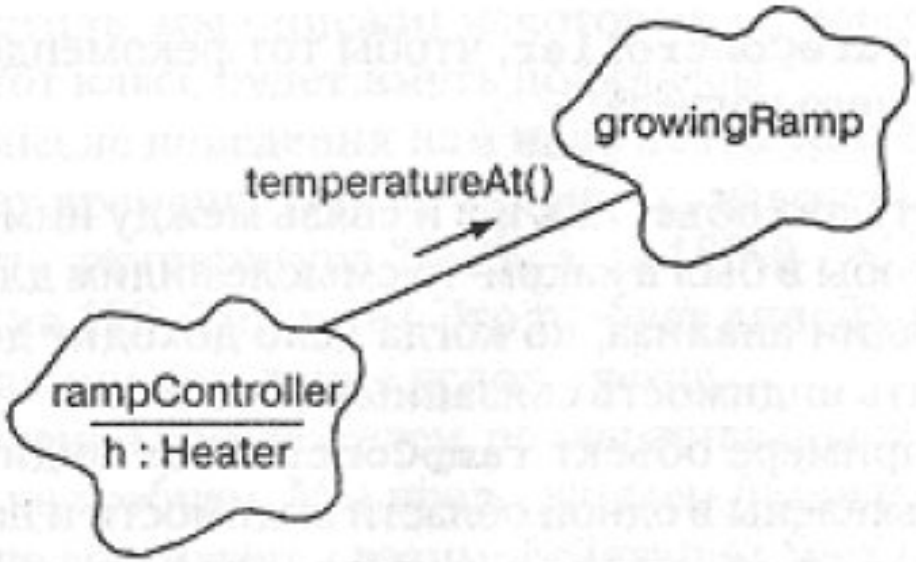
Самолет состоит из крыльев, двигателей, шасси и т.п.

Отношения акционера с его акциями - это агрегация, которая не предусматривает физического включения. Акционер владеет акциями

Агрегация

описывает отношения целого и части, приводящие к соответствующей иерархии объектов. Через целое (агрегацию) можно получить доступ к его частям (атрибутам).

ЧАСТЬ



ЦЕЛОЕ

Объект **rampController** имеет:

- связь с объектом **growingRamp**
- атрибут h класса **Heater** (нагреватель). **rampController** - целое, а h- его часть. h - часть состояния **rampController**. Через **rampController**, можно найти соответствующий нагреватель.

Но по h нельзя найти содержащий его объект (называемый также его контейнером), если только сведения о нем не являются случайно частью состояния h.

Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом. Связи иногда предпочтительнее, так как они слабее и менее ограничительны.

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

Пример. Можно добавить в спецификацию класса **TemperatureController** описание нагревателя:

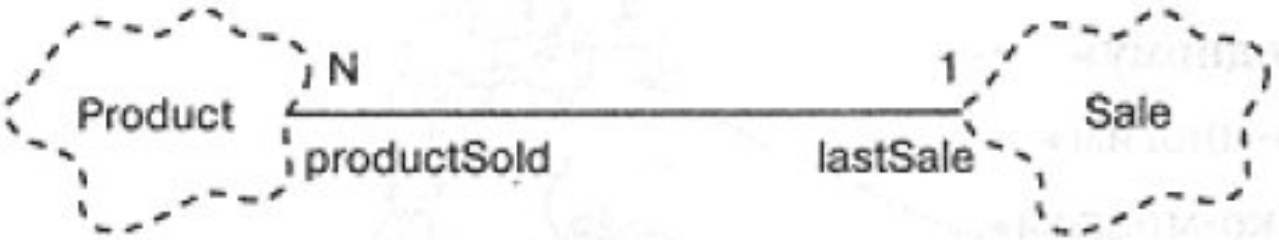
- **Heater h;**
- После этого каждый объект **TemperatureController** будет иметь свой нагреватель. Нужно инициализировать нагреватель при создании нового контроллера, так как сам этот класс не предусматривает конструктора по умолчанию. Можно определить конструктор класса **TemperatureController** следующим образом:
- **TemperatureController::TemperatureController(Location 1) : h(1) {}**

Отношения между классами

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

- ассоциация (is-a отношение)
- наследование (альтернативой наследованию является делегирование)
- агрегация (has-a отношение)
- использование
- инстанцирование
- метакласс

Ассоциация - смысловая связь



Три случая мощности ассоциации:

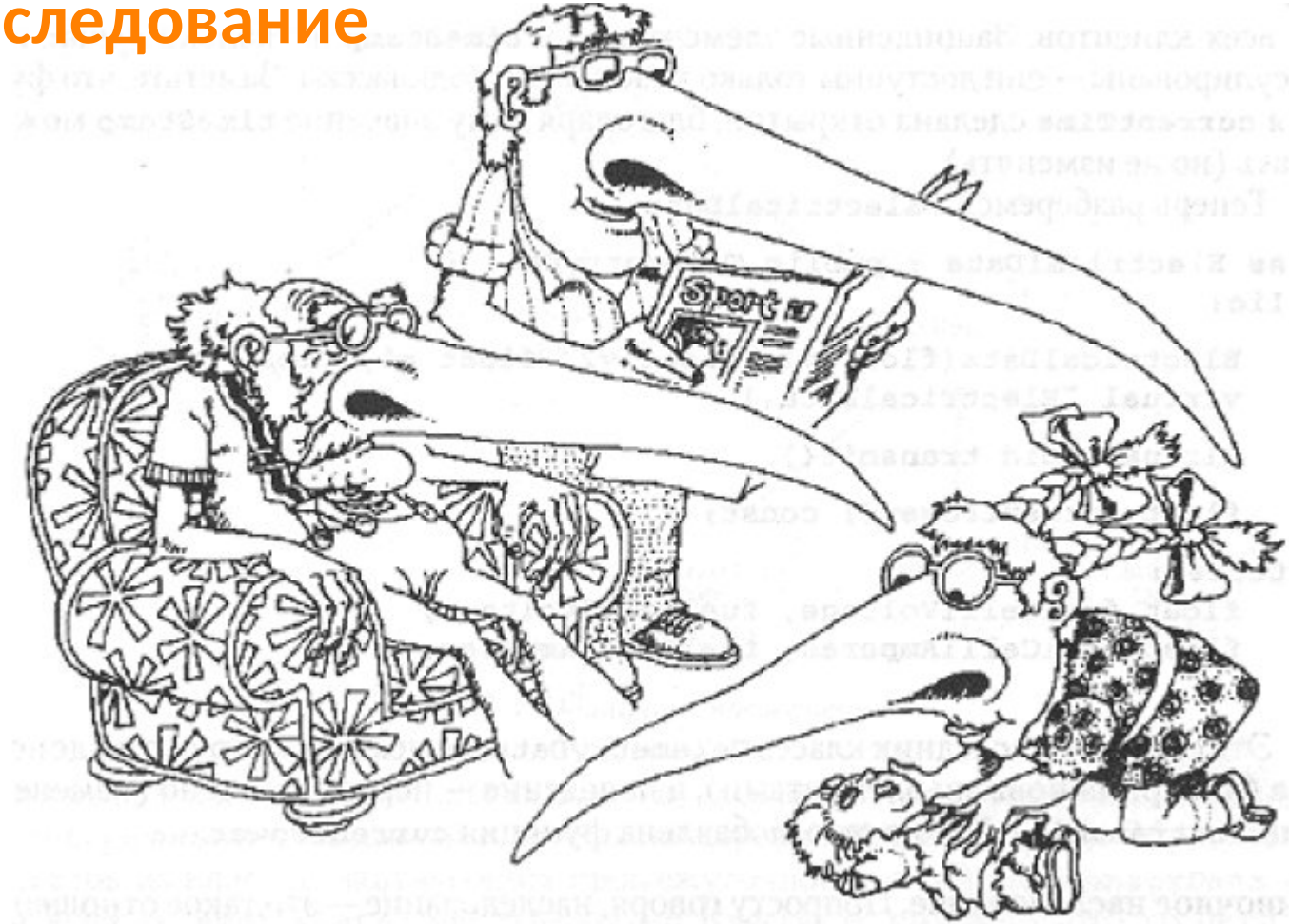
- "один-к-одному"
- "один-ко-многим"
- "многие-ко-многим"

Пример. Автоматизация розничной торговой точки: две абстракции - товары и продажи. Класс **Product** – то, что продано в некоторой сделке, класс **Sale** - сама сделка, в которой продано несколько товаров. Ассоциация: зная товар, можно выйти на сделку, в которой он был продан, а зная сделку - найти, что было продано

```
class Product;
class Sale;
class Product {
public:
...
protected:
Sale* lastSale;
};
class Sale {
public:
...
protected:
Product** productSold;
};
```

Ассоциация вида "один-ко-многим": каждый экземпляр товара относится только к одной последней продаже, каждый экземпляр **Sale** может указывать на совокупность проданных товаров

Наследование



Наследование

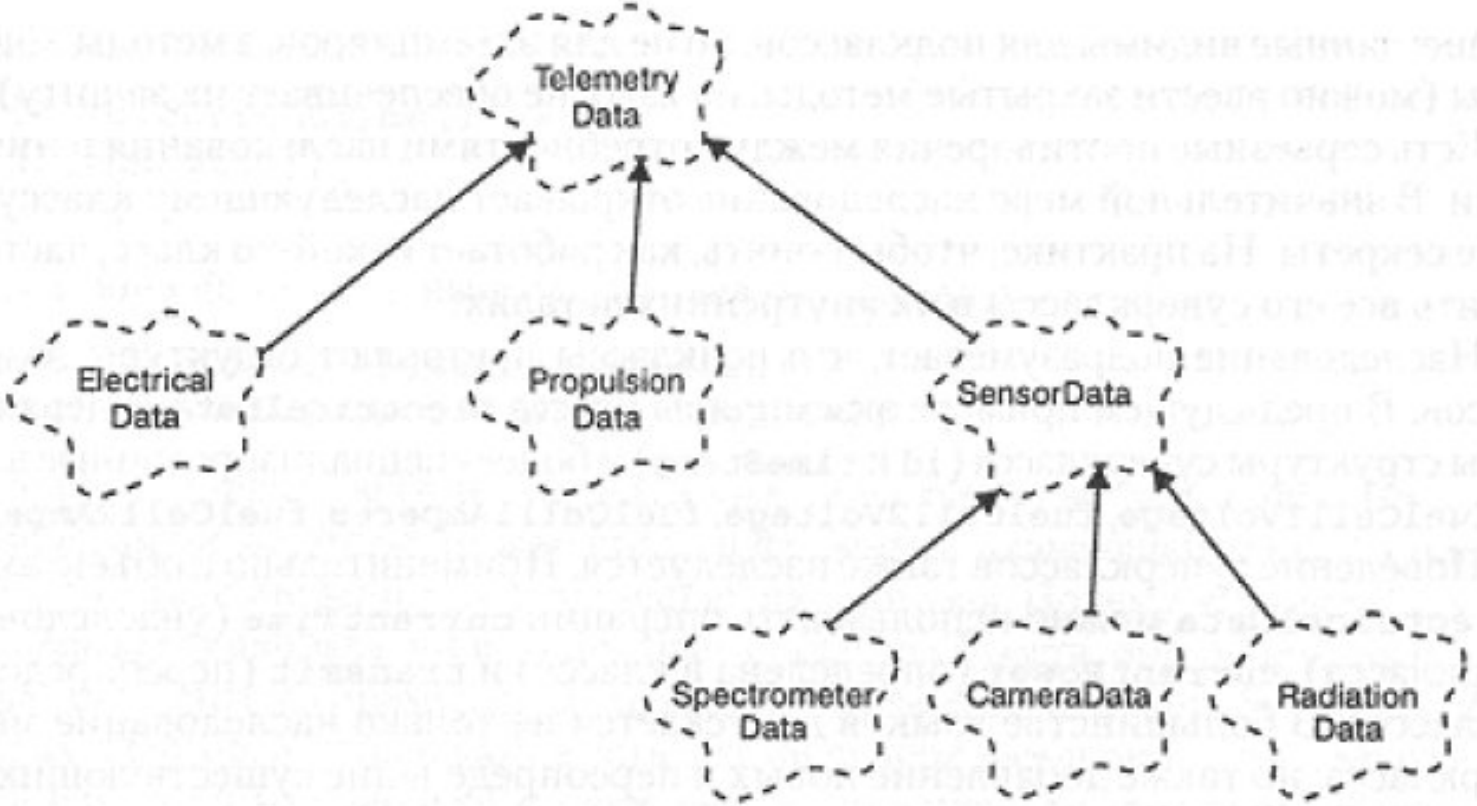
- отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов

Класс, структура и поведение которого наследуются, называется суперклассом. **TelemetryData**. является суперклассом для **ElectricalData**. Производный от суперкласса класс называется подклассом. Наследование устанавливает между классами иерархию общего и частного. **ElectricalData** является более специализированным классом более общего **TelemetryData**

Наследование

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

```
class ElectricalData : public
TelemetryData {
public:
ElectricalData(float vl, float v2,
float al, float a2);
virtual ~ElectricalData();
virtual void. transmit();
float currentPower () const;
protected:
float fuelCelllVoltage,
fuelCell2Voltage;
float fuelCelllAmperes,
fuelCell2Amperes;
}; };
```



Одиночное наследование

Множественное наследование

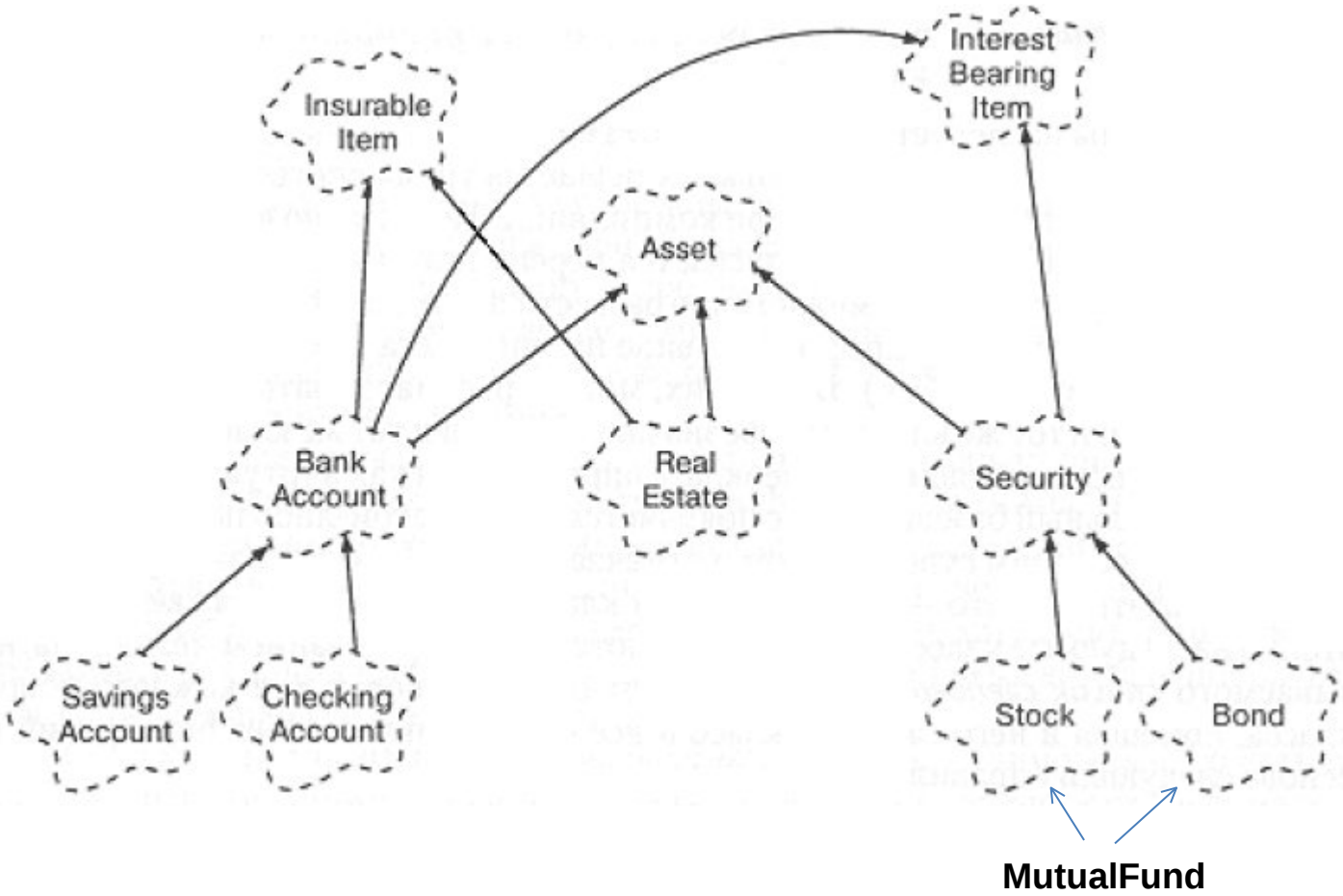
- класс наследует от нескольких суперклассов

• Проблемы: конфликты имен, ромбическое наследование

• Решение в C++: виртуальные базовые классы

• Альтернатива: интерфейсы (как в Java)

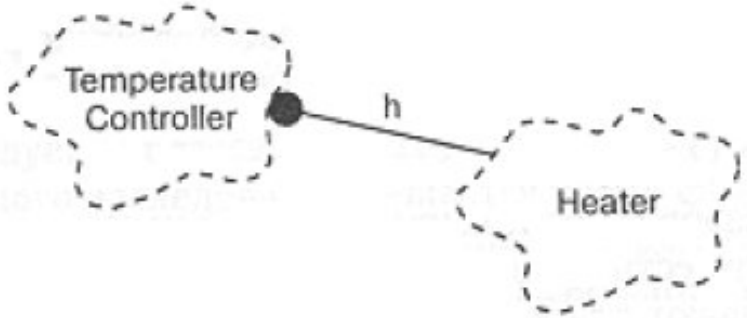
Множественное наследование



Наследование классов

- Механизм повторного использования кода
- Специализация общих абстракций
- Иерархия "обобщение-специализация"
- Производный класс наследует структуру и поведение
- Может добавлять, модифицировать, скрывать унаследованное

Агрегация

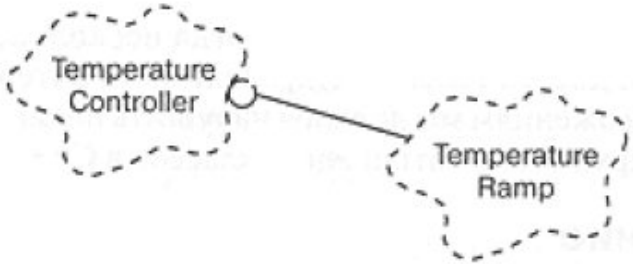



```
class TemperatureController {
public:
TemperatureController(Location) ;
~TemratureController();
void process(const TemperatureRamp&);
Minute schedule(const TemperatureRamp&)
const;
private:
Heater h;
};
```

Отношение использования

TemperatureController и **TemperatureRamp**.

```
class TemperatureController {
public:
TemperatureController(Location) ;
~TemperatureController();
void process(const TemperatureRamp&);
Minute schedule(const TemperatureRamp&) const;
private:
Heater h;
};
```



Класс **TemperatureRamp** - часть сигнатуры функции-члена process;
Класс **TemperatureController** пользуется услугами класса **TemperatureRamp**.

Инстанцирование

```
Template<class Item>
class Queue {
public:
Queue() ;
Queue(const Queue<Item>&);
virtual ~Queue();
virtual Queue<Item>& operator=(const
Queue<Item>&);
virtual int operator==(const Queue<Item>&) const;
int operator!=(const Queue<Item>&) const;
virtual void clear();
virtual void append(const Item&);
virtual void pop();
virtual void remove(int at);
virtual int length() const;
virtual int isEmpty() const;
virtual const Item& front() const;
virtual int location(const void*);
protected:
…
};
```

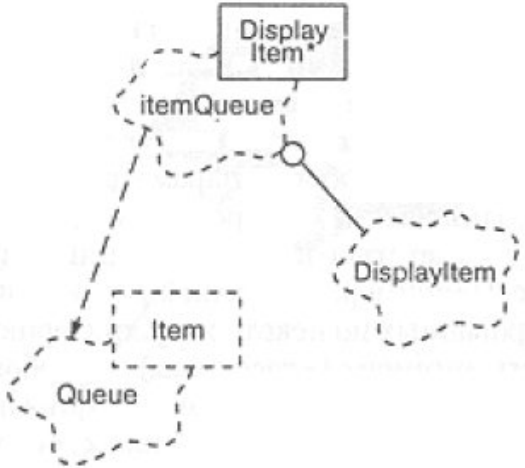
- Объекты помещаются в очередь и достаются из нее при помощи класса Item, объявленного в шаблоне
- Параметризованный класс не может иметь экземпляров, пока он не будет инстанцирован

Queue intQueue;
Queue itemQueue;

Объекты intQueue и itemQueue - экземпляры различных классов, которые не имеют общего суперкласса. Они получены из одного параметризованного класса Queue

По правилам C++ нельзя будет поместить в очередь или извлечь из нее что-либо кроме, соответственно, целых чисел и разновидностей DisplayItem.

Отношения между параметризованным классом **Queue**, его инстанцированием для класса **DisplayItem** и экземпляром **itemQueue**

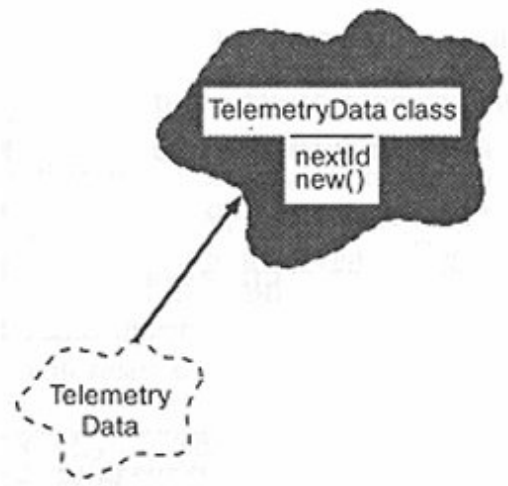


Инстанцирование

Метакласс

- попытка обращения с классом, как с объектом
- Класс класса - метакласс

в C++ метаклассов нет, но . C++ имеет средства поддержки и переменных класса, и операций метакласса.



Часть 3

Взаимосвязь классов и объектов

Взаимосвязь классов и объектов

- Класс описывает, объект существует
- Объект — экземпляр (instance) класса
- Инстанцирование: создание объектов из класса
- Статические члены класса: общие для всех экземпляров
- Динамические члены: уникальные для каждого объекта

Пример: создание объектов

```
// Определение класса
class TemperatureSensor { ... };

// Создание объектов (экземпляров)
TemperatureSensor sensor1(Location(1));
TemperatureSensor sensor2(Location(2));
TemperatureSensor* sensor3 = new TemperatureSensor(Location(3));
```

- Каждый объект имеет собственное состояние
- Общее поведение определяется классом
- Объекты могут быть созданы статически или динамически

Качество классов и объектов

Критерии хорошего дизайна классов

- Связность (cohesion): насколько класс делает одно дело
- Сцепление (coupling): зависимости между классами
- Полнота и примитивность операций
- Существенность: абстракция должна быть значимой

Принципы хорошего дизайна

- Высокая связность, низкое сцепление
- Минимальный и полный интерфейс
- Инкапсуляция изменяющихся аспектов
- Наследование для специализации, а не для повторного использования
- Полиморфизм вместо проверки типов

Пример класса Queue

```
class Queue {
  public:
    Queue();
    Queue(const Queue&);
    virtual ~Queue();
    virtual Queue& operator=(const Queue&);
    virtual int operator==(const Queue&) const;
    virtual void clear();
    virtual void append(const void*);
    virtual void pop();
    virtual int length() const;
  protected:
    ...
};
```

- Полный интерфейс очереди
- Виртуальные функции для переопределения
- Шаблонный метод для произвольных типов (void*)

Антипаттерны дизайна классов

- Божественный объект: слишком много ответственности
- Анемичная модель: данные без поведения (есть приватные поля, get, set, но нет логики)
- Наследование для реализации вместо специализации (создание наследника вместо специализации)
- Нарушение инкапсуляции: геттеры/сеттеры для всего
- Жесткие зависимости вместо слабой связанности

Выводы

- Объекты: состояние + поведение + идентичность
- Классы: шаблоны для создания объектов
- Отношения: наследование, агрегация, ассоциации
- Качество дизайна: связность, сцепление, полнота

Рекомендации по применению ООП

1. Выявить объекты в предментной области
2. Сгруппировать похожие объекты в классы
3. Выявить связи между объектами и классами
4. Выполнить проектирование интерфейсов
5. Протестировать абстракции на соответствие реальности

Объектно-ориентированное программирование:

моделирование реального мира через взаимодействующие объекты

- Объектная модель как ответ на сложность ПО
- От алгоритмической к объектной декомпозиции
- ООА, ООД и ООП как единый подход