# Object Oriented Programming

Classes and Objects

(object orientation)

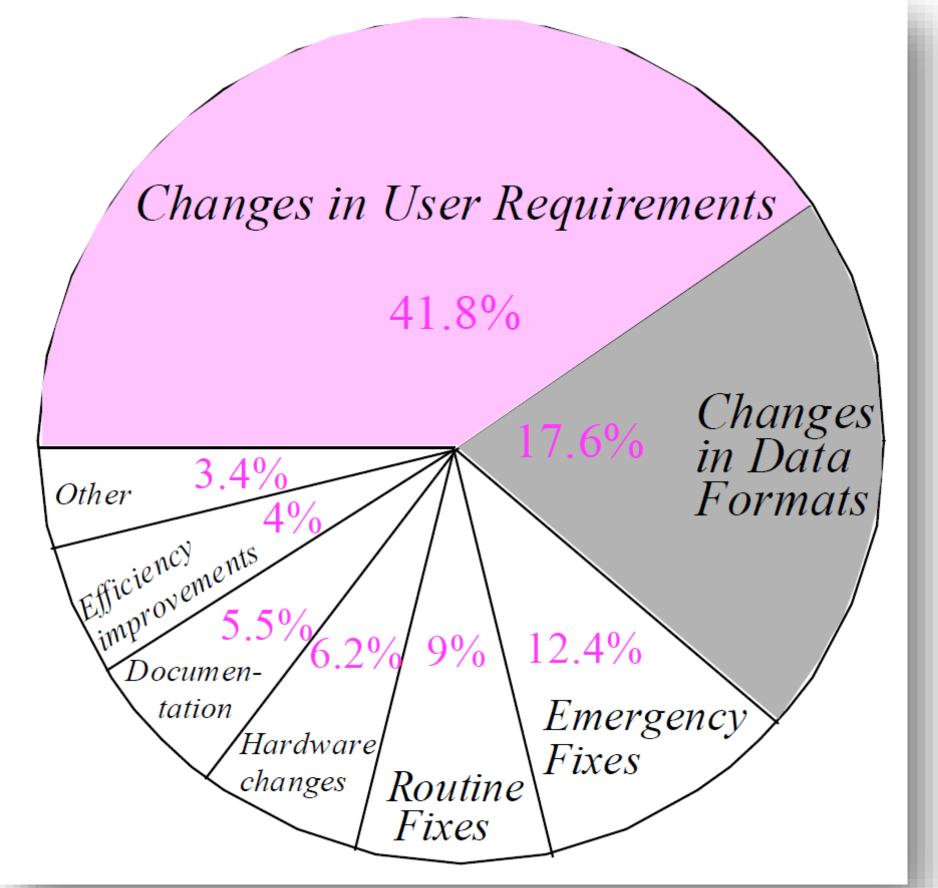
2023/24

# Lecture Outline

- Concept of object-oriented design (why we need objects)
- Classes, objects…
- Example

# Why do we need objects?

# Software Maintenance




- Study of Lientz and Swanson (1980)
- 487 information systems
- **How much effort does it require?**

# History?

- *Simula 67* language (Ole-Johan Dahl a Kristen Nygaard, Norwegian Computing Center, 1960+)
  - classes and instances (objects)
  - automatic object destruction (garbage collection)
- *Smalltalk* langauge (Xerox PARC, Alan Kay a další, 1970+)
  - “object-oriented programming“ as a new term
  - using objects and messages (message passing and processing)

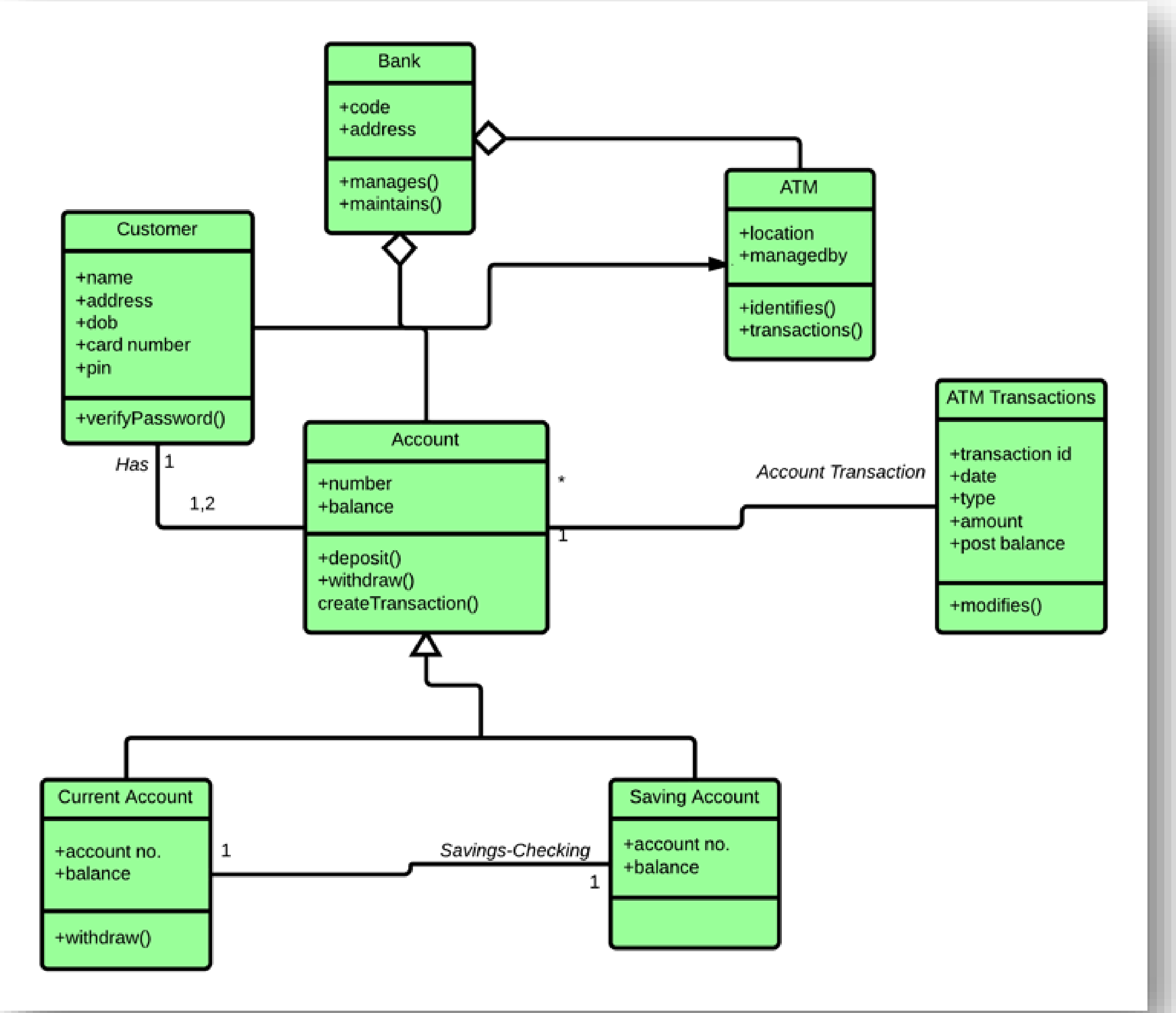
# Object Orientation…

- Object-oriented techniques help to write software with better maintainability
    - Method and language
    - Implementation and environment
    - Libraries

# Method and Language

- It is not just programming language and how to use it.
- It is also a way of thinking and expressing ...
- ... and also about records in textual or graphical form.

# Domain Model



# Implementation and Language

- Support of the development
  - Features and efficiency of development tools
  - Tools supporting the deployment of new versions
  - Tools to support documentation

# Libraries

- Object-oriented approaches rely heavily on reusability.
- To support development by using previously implemented solutions (libraries)
- It is also to support the creation and management of new custom libraries

# Should we be dogmatic?

- The object-oriented approach is an essential tool for software development. However:
  - There are various programming languages with varying degrees of support for object-oriented programming techniques (OOP)
  - Not everyone needs all the features that OOP offers
  - Object orientation may be just one factor in the successful development, and therefore, it should be considered comprehensively

# Method and Language

- Classes
- Classes as modules
- Classes as types
- Message passing (feature call)
- Information hiding
- The static type checking
- Genericity
- Inheritance, redefinition, polymorphism and dynamic binding
- Memory management and garbage collection

# Class

# Classes

> The method and the language should have the notion of class as their central concept.

- The object-oriented approach is based on the term class.
- The class can be seen as part of the software, which describes the abstract data type and its implementation.
- As the abstract data type, we understand a set of objects with a common behavior represented by a list of operations that objects can operate.

# Classes as Modules

> Classes should be the only modules.

- The OOP is mainly about the software structure (architecture); its priority is modularity.
- Classes not only describe the types of objects; they must also be modular units.
- In pure object-oriented programs should not be other separate units than classes (e.g., functions).

# Classes as Types

> Every type should be based on a class.

- In pure object-oriented languages and programs should not be other types than the classes.

- This principle also can be applied to the system types such as INT or FLOAT.

# Message Passing

> Feature call should be the primary computational mechanism.

- *Message Passing* (feature class), *feature-based computation* - a computational mechanism.
  - A named message (with parameters) is sent to an object (an instance of a class).
  - *aPerson-> ChangeLastName ("Smith")*
  - Whoever sends a message (requesting execution of operations with certain arguments) is a CLIENT of the class.

# Information Hiding

> It should be possible for the author of a class to specify that a feature is available to all clients, to no client, or to specified clients.

- For the client, only the operations (methods) that describe the external behavior of objects are essential.
- Details of implementation should be hidden (data + private operations).
- If a client needs to obtain information about the state (data) of an object, it is possible only by sending a message.

# Static Binding and Type Checking

> A well-defined type system should, by enforcing a number of type declaration and compatibility rules, guarantee the run-time type safety of the systems it accepts.

- Each entity in the program (e.g., a variable) must have a defined type.
- Any request for an object (the message) must correspond to the operation (method) that the class provides.

# Genericity

> It should be possible to write classes with formal generic parameters representing arbitrary types.

- It is necessary to have classes that can work with a type that is not known in advance.

- As an example, we can need lists to store objects of different classes (types).

# Inheritance and Redefinition

It should be possible to define a class as inheriting from another.

It should be possible to redefine the specification, signature and implementation of an inherited feature.

- Inheritance enables to build of a new class based on an existing one. The basic idea is an extension of the original class by new features.

- In the context of inheritance, we can also require changing some features of the original class.

# Polymorphism and Dynamic Binding

> It should be possible to attach entities (names in the software texts representing run-time objects) to run-time objects of various possible types, under the control of the inheritance-based type system.

> Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call.

- Sometimes we need a single object featured in a different context in a different role.
- A role means a different behavior, which may vary in time.

# Memory Management and Garbage Collection

> The language should make safe automatic memory management possible, and the implementation should provide an automatic memory manager taking care of garbage collection.

- In large programs, many new objects are constructed and destructed over time in different contexts.
- There is a problem with managing the life cycle of these objects manually.
- The support of object destruction must be done automatically.

# Example

# Declaration

The *constructor* initializes the object (puts the data into the memory that the object uses)

The *destructor* deletes object data (frees the memory occupied by the object)

```
#include <iostream>
using namespace std;

class KeyValue
{
private:
    int key;
    double value;

public:
    KeyValue(int k, double v);
    int GetKey();
    double GetValue();
};
```

```
class KeyValues
{
private:
    KeyValue** keyValues;
    int count;

public:
    KeyValues(int n);
    ~KeyValues();
    KeyValue* CreateObject(int k, double v);
    KeyValue* SearchObject(int key);
    int Count();
};
```

# Using the Class

Using the keyword *new* ensures the object's creation (allocates memory for data - "flat" part - of the object).

```
int main()
{
    int N = 5;
    KeyValues* myKeyValues = new KeyValues(N);

    KeyValue* myKeyValue = myKeyValues->CreateObject(0, 0.5);
    cout << myKeyValue->GetValue() << endl;

    for (int i = 1; i < N; i++)
    {
        myKeyValues->CreateObject(i, i + 0.5);
    }
    cout << myKeyValues->SearchObject(4)->GetValue() << endl;

    delete myKeyValues;

    //cout << myKeyValue->GetKey() << endl;

    getchar();
    return 0;
}
```

# Result

```
0.5
4.5
```

# Class definition (implementation)

```cpp
KeyValues::KeyValues(int n)
{
    this->keyValues = new KeyValue*[n];
    this->count = 0;
}

KeyValues::~KeyValues()
{
    for (int i = 0; i < this->count; i++)
    {
        delete this->keyValues[i];
    }

    delete[] this->keyValues;
}
```

```cpp
int KeyValues::Count()
{
    return this->count;
}
```

```cpp
KeyValue* KeyValues::CreateObject(int k, double v)
{
    KeyValue *newObject = new KeyValue(k, v);

    this->keyValues[this->count] = newObject;
    this->count += 1;

    return newObject;
}

KeyValue* KeyValues::SearchObject(int k)
{
    for (int i = 0; i < this->count; i++)
    {
        if (this->keyValues[i]->GetKey() == k)
        {
            return this->keyValues[i];
        }
    }

    return nullptr;
}
```

# Seminar assignments

- Implement the example from the lecture; add the *KeyValues* class *KeyValue * RemoveObject (int k)* method that removes the object with that key and returns the pointer to this object.

- Implement an *Invoice* class, which will include *invoice number*, an object of a *Person* class (with the *name* and *address*), and an *array* of objects (pointers) of an *InvoiceItem* class (with *title* and *price*). Design and implement *constructor* and *destructor* and other necessary methods. The invoice will have a method (function) that calculates and returns the *total price*.

# Seminar Questions

- What are the main reasons for software changes?
- What are the main factors influencing object orientation?
- Explain what the object-oriented method and language are.
- Explain the support of object-oriented implementation.
- Explain what reusability is (using and building libraries).
- Explain the concepts of class and object and use the correct terminology.
- Explain class properties regarding modularity.
- Explain the principle of encapsulation in OOP.
- Explain the principle of message passing.
- Explain the principles of the declaration and the definition of a simple class in C ++.

# Sources

- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1997. [17-36]