OBJECT-ORIENTED PROGRAMMING
IN C++

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
https://akademi.itu.edu.tr/en/buzluca
http://www.buzluca.info

INTRODUCTION

Main Objectives of the Course :

- To introduce **Object-Oriented Programming** and **Generic Programming**
- To show how to use these programming schemes with the C++ programming language to build "**good**" (high-quality) programs.

Need for high-quality design and good programming methods:

Today, almost every electronic device includes a computer system controlled by software.

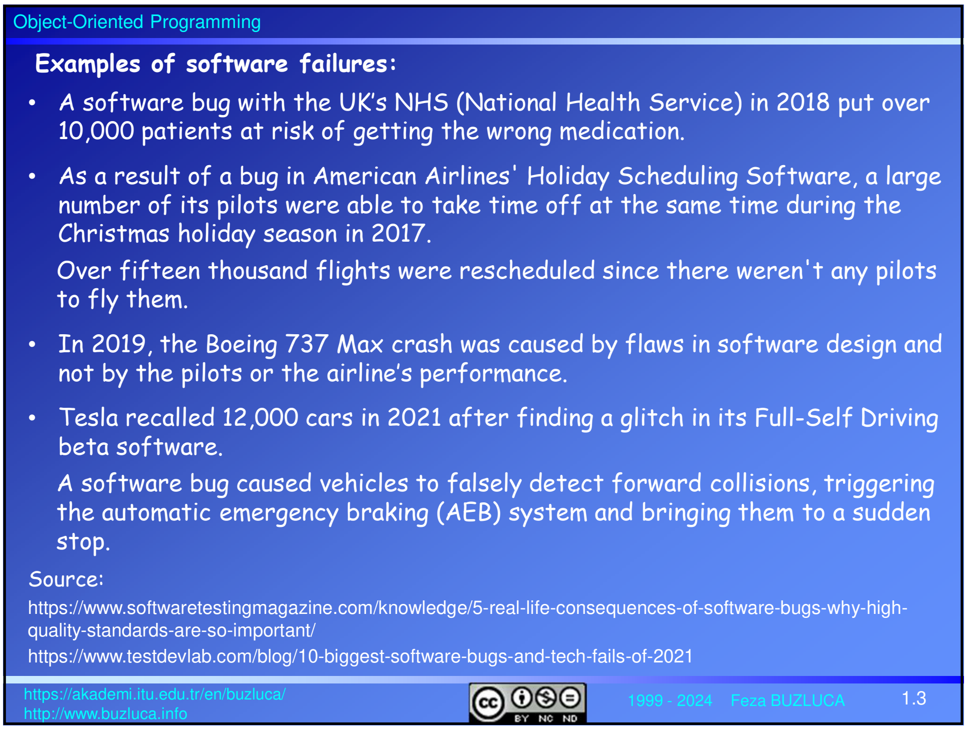
Software plays a vital role in our daily lives.

Problems:

- Software project costs (especially maintenance costs) are high.

  Maintenance: Changes (requirement changes or bug fixes) and extensions must be made to the software system after it has been delivered to the customer.

- Software errors may cause financial losses, loss of lives and jobs.



Examples of software failures:

- A software bug with the UK's NHS (National Health Service) in 2018 put over 10,000 patients at risk of getting the wrong medication.
- As a result of a bug in American Airlines' Holiday Scheduling Software, a large number of its pilots were able to take time off at the same time during the Christmas holiday season in 2017.

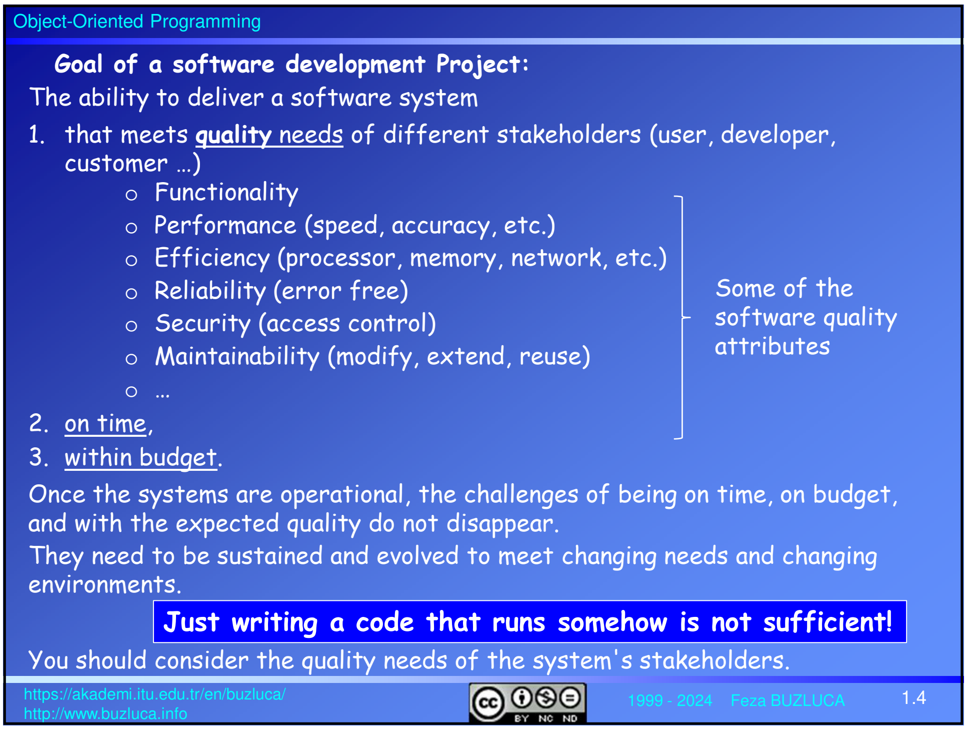
  Over fifteen thousand flights were rescheduled since there weren't any pilots to fly them.
- In 2019, the Boeing 737 Max crash was caused by flaws in software design and not by the pilots or the airline's performance.
- Tesla recalled 12,000 cars in 2021 after finding a glitch in its Full-Self Driving beta software.

  A software bug caused vehicles to falsely detect forward collisions, triggering the automatic emergency braking (AEB) system and bringing them to a sudden stop.

Source:

https://www.softwaretestingmagazine.com/knowledge/5-real-life-consequences-of-software-bugs-why-high-quality-standards-are-so-important/

https://www.testdevlab.com/blog/10-biggest-software-bugs-and-tech-fails-of-2021



Goal of a software development Project:

The ability to deliver a software system

1. that meets **quality needs** of different stakeholders (user, developer, customer …)
   - Functionality
   - Performance (speed, accuracy, etc.)
   - Efficiency (processor, memory, network, etc.)
   - Reliability (error free)
   - Security (access control)
   - Maintainability (modify, extend, reuse)
   - …

   Some of the software quality attributes
2. on time,
3. within budget.

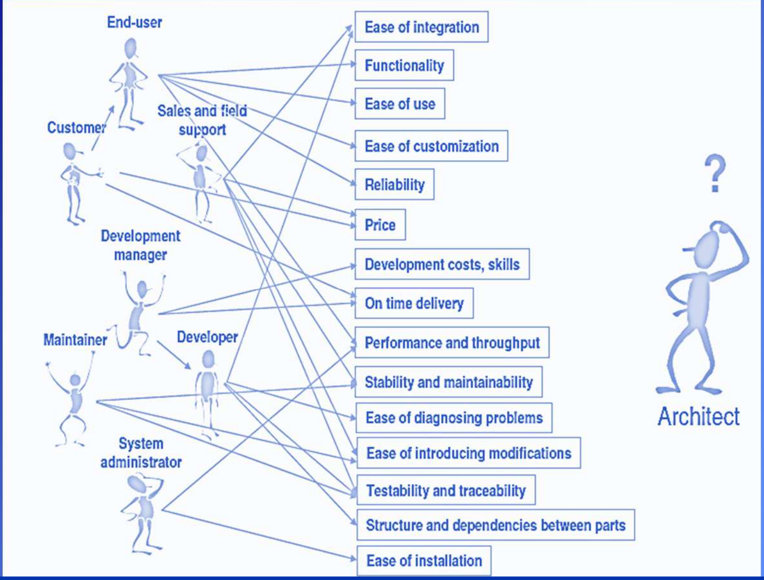
Once the systems are operational, the challenges of being on time, on budget, and with the expected quality do not disappear.

They need to be sustained and evolved to meet changing needs and changing environments.

Just writing a code that runs somehow is not sufficient!

You should consider the quality needs of the system's stakeholders.

Expectations of different stakeholders (Quality needs):



Source: D. Falessi, G. Cantone, R. Kazman, and P. Kruchten, "Decision-making techniques for software architecture design," *ACM Computing Surveys*, vol. 43, pp. 1-28, Oct. 2011.

Quality characteristics of a software system

ISO (the International Organization for Standardization) and **IEC** (the International Electrotechnical Commission) prepared standards for quality models.

You may find definitions of the quality attributes of a software system in the following standard.

ISO/IEC 25010: *Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*

This standard includes two quality models.

A) Quality in use model:

This is the external quality of the system; the impact on stakeholders (customers, direct and indirect users, etc.) in specific contexts of use.

B) Product Quality:

These characteristics relate to the software development team.

Details of the quality models are out of the scope of this course.

They will be covered in BLG 468E "Object-Oriented Modeling and Design" (8th semester) and BLG 625 "Software Design Quality " (graduate) courses.

This course will give only a brief insight into a software system's quality attributes that must always be considered during software development.

Quality Attributes of a Software (External and Internal)

External

- A program must do its job correctly (effectiveness).
- A program must perform as fast as necessary (Time constraints).
- It must not waste system resources (processor time, memory, disk capacity, network capacity) too much (efficiency).
- It must be reliable (trustful).
- It must be useful, usable, and have enough documentation (easy to learn and use).
- It must be easy to update (**extend**, **adapt**) the program (flexibility).

User

Internal

- It must be functionally complete and correct.
- It must be efficient (time behavior, resource utilization, capacity).
- Source code must be readable and understandable (comments, documentation).
- It must be easy to **extend** and **update** (change) the program according to new requirements and **adapt** it to new environments.
- It must be easy to test the program to find and correct errors.
- Modules of the program must be reusable in further projects.

Software developer

While designing and coding a program (***and learning a programming language***), these quality attributes must always be considered.

Why Object-Oriented Technology?

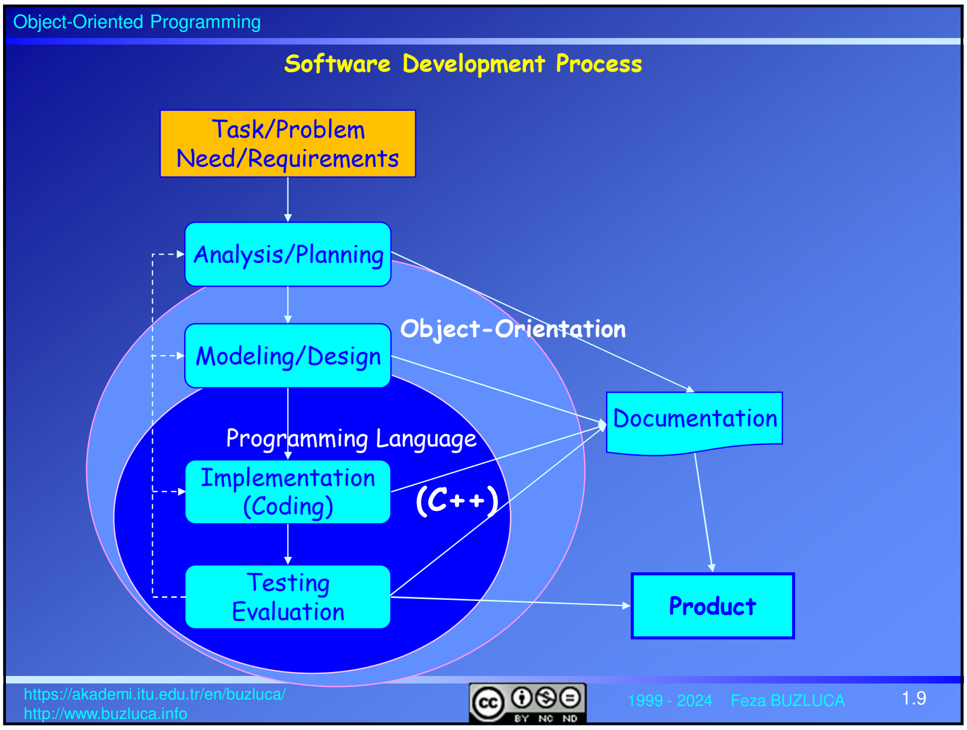
Expectations are,

- Reducing the effort, complexity, and **cost of development**.
- Reducing the **cost of maintenance** (finding bugs, correcting them, improving the system).
- Reducing the **cost of extending the system** (adding new features).
- Reducing the effort to **adapt an existing system** (quicker reaction to changes in the business environment) (flexibility).
- Reducing the effort to **use existing modules in a new project** (reusability).
- Increasing the **reliability** of the system (fewer failures.)

Object-oriented programming technique enables programmers to build high-quality programs.

While you design and code a program, **you must consider these expectations**.

If there are multiple options when writing a program, you should choose one that meets these expectations.



Software Development Process

Task/Problem
Need/Requirements

Analysis/Planning

Modeling/Design

Object-Orientation

Programming Language

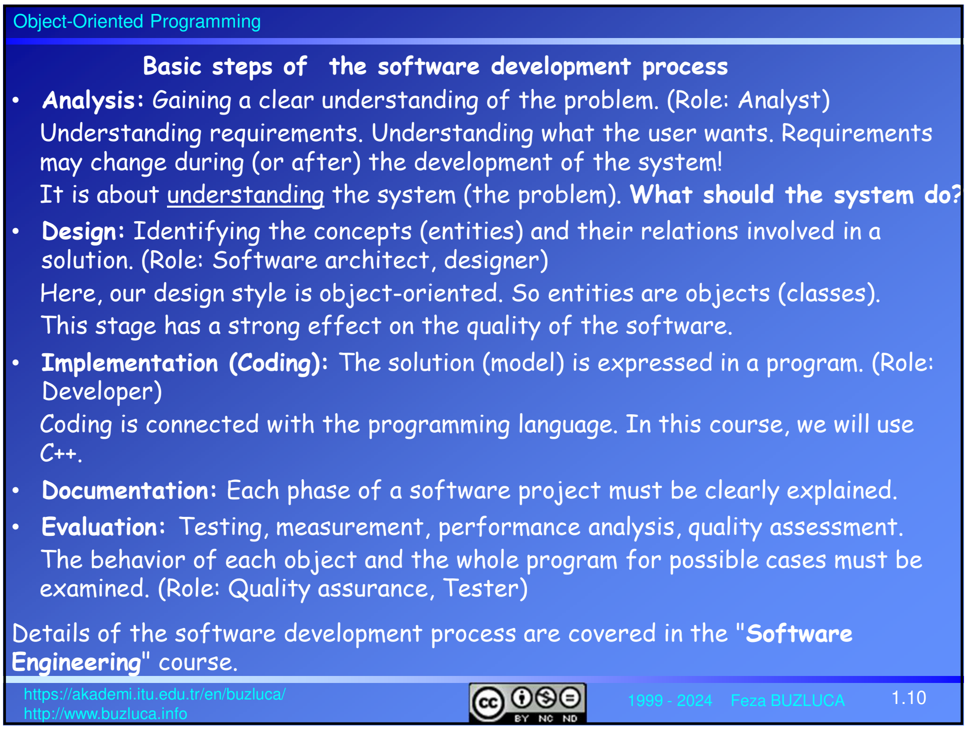
Implementation
(Coding)

(C++)

Testing
Evaluation

Documentation

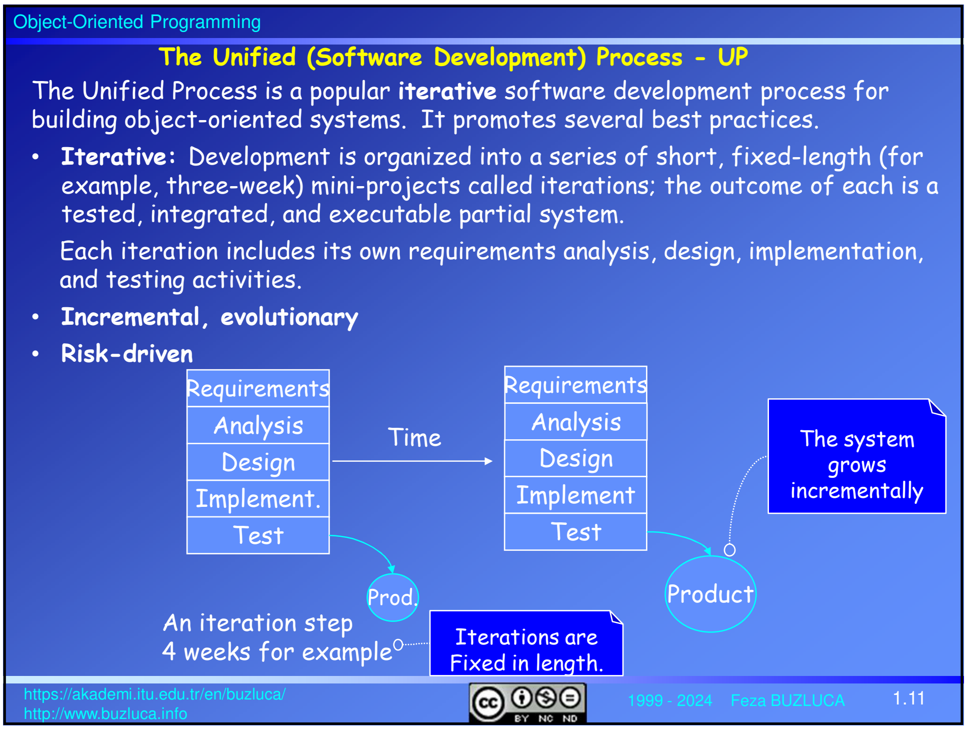
Product



Basic steps of the software development process

- **Analysis:** Gaining a clear understanding of the problem. (Role: Analyst)
  Understanding requirements. Understanding what the user wants. Requirements may change during (or after) the development of the system!
  It is about <u>understanding</u> the system (the problem). **What should the system do?**
- **Design:** Identifying the concepts (entities) and their relations involved in a solution. (Role: Software architect, designer)
  Here, our design style is object-oriented. So entities are objects (classes).
  This stage has a strong effect on the quality of the software.
- **Implementation (Coding):** The solution (model) is expressed in a program. (Role: Developer)
  Coding is connected with the programming language. In this course, we will use C++.
- **Documentation:** Each phase of a software project must be clearly explained.
- **Evaluation:** Testing, measurement, performance analysis, quality assessment.
  The behavior of each object and the whole program for possible cases must be examined. (Role: Quality assurance, Tester)

Details of the software development process are covered in the "**Software Engineering**" course.



The Unified (Software Development) Process - UP

The Unified Process is a popular **iterative** software development process for building object-oriented systems. It promotes several best practices.

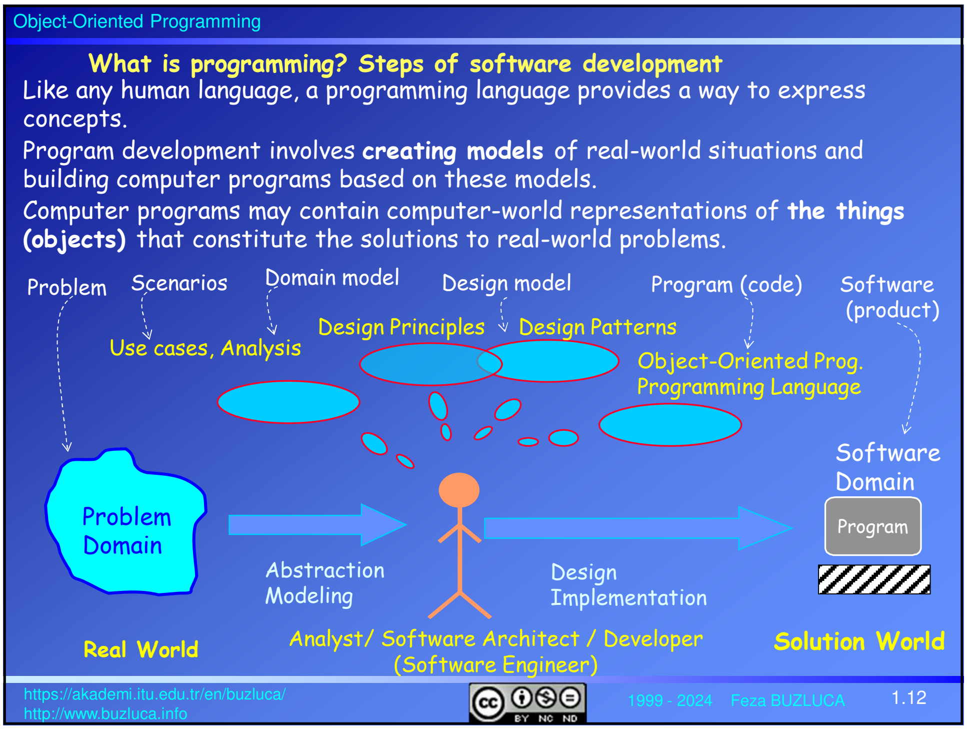
- **Iterative:** Development is organized into a series of short, fixed-length (for example, three-week) mini-projects called iterations; the outcome of each is a tested, integrated, and executable partial system.

  Each iteration includes its own requirements analysis, design, implementation, and testing activities.
- **Incremental, evolutionary**
- **Risk-driven**

The system grows incrementally

An iteration step
4 weeks for example

Iterations are Fixed in length.



What is programming? Steps of software development

Like any human language, a programming language provides a way to express concepts.

Program development involves **creating models** of real-world situations and building computer programs based on these models.

Computer programs may contain computer-world representations of **the things (objects)** that constitute the solutions to real-world problems.

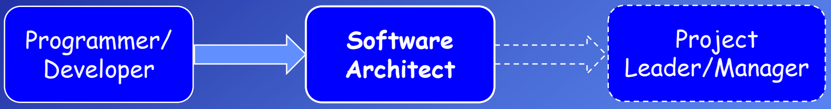
A possible road in your professional life

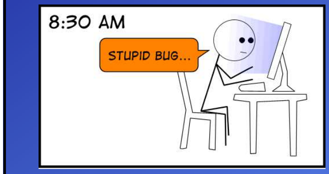
If you will work in the world of software development

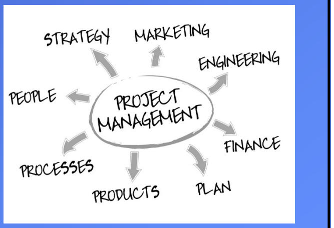
Writes the code.

Designs the architecture.
Coaches the team.
Decides.

Management duties.
Not only about software







Source:
http://www.smashingapps.com/

Source:
http://www.planetgeek.ch

Source:
http://www.businessadministrationinformation.com/

Learning a Programming Language

- Knowledge about a programming language's grammar rules (syntax) is not enough to write “good” programs.
- The essential thing to do when learning to program is to focus on concepts (**design and programming techniques**) and not get lost in language-technical details.
- Rather than the programming language's rules, the programming scheme must be understood.

  Understanding design techniques comes with time and practice.
- Learn and use design principles and design patterns.
- Always consider quality characteristics (understandability, flexibility, ...).

Why C++?

The main objective of this course is not to teach a programming language. However, examples are given in C++.

Properties of the C++ programming language:

- C++ supports object-oriented and generic programming.
- Performance (especially speed) of programs written with C++ is high.
- It is helpful in low-level programming environments where direct control of hardware is necessary.

  Embedded systems and compilers are created with the help of C++.
- C++ gives the user control over memory management (also increases the programmer's responsibility "*with authority comes responsibility*").
- C++ is used by hundreds of thousands of programmers in every application domain.
  - Hundreds of libraries support this use,
  - hundreds of textbooks, several technical journals, and many conferences.
- C++ programmers can quickly adapt to other object-oriented programming languages such as Java or C#.

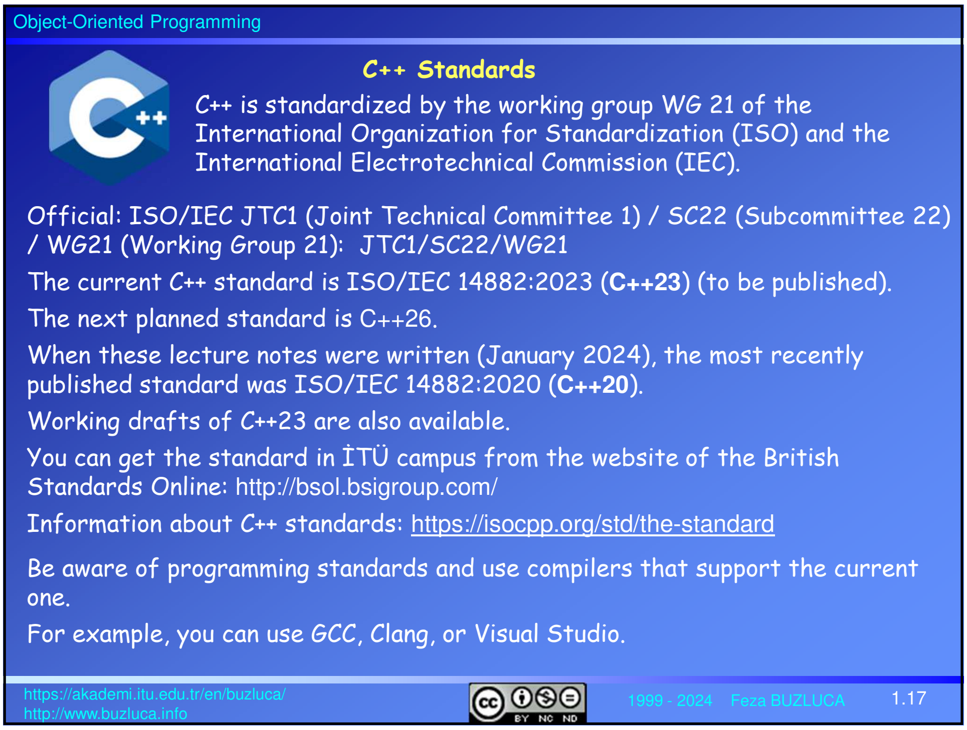
The applications domain of C++:

- Game (engine) development: Speed and control over hardware are crucial. Examples: Fortnite and Unreal Engine
- Graphics and user interface programs
- Systems programming: Operating systems, device drivers. Here, direct manipulation of hardware under real-time constraints is essential.
- High-performance applications: Scientific computing and financial modeling.
- Embedded systems: For example, systems for cars and medical devices.
  It is possible to implement relatively small and efficient programs that can run on limited hardware resources.

Examples of applications written in C++:

- Apple's Mac OS X,
- Adobe Systems,
- Backend services of Facebook,
- Google's Chrome browser,
- Microsoft Windows operating systems, MS Office, Visual Studio
- Mozilla Firefox, Thunderbird,
- MySQL

are written in part or in their entirety with C++.



C++ Standards

C++ is standardized by the working group WG 21 of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).

Official: ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21): JTC1/SC22/WG21

The current C++ standard is ISO/IEC 14882:2023 (**C++23**) (to be published).

The next planned standard is C++26.

When these lecture notes were written (January 2024), the most recently published standard was ISO/IEC 14882:2020 (**C++20**).

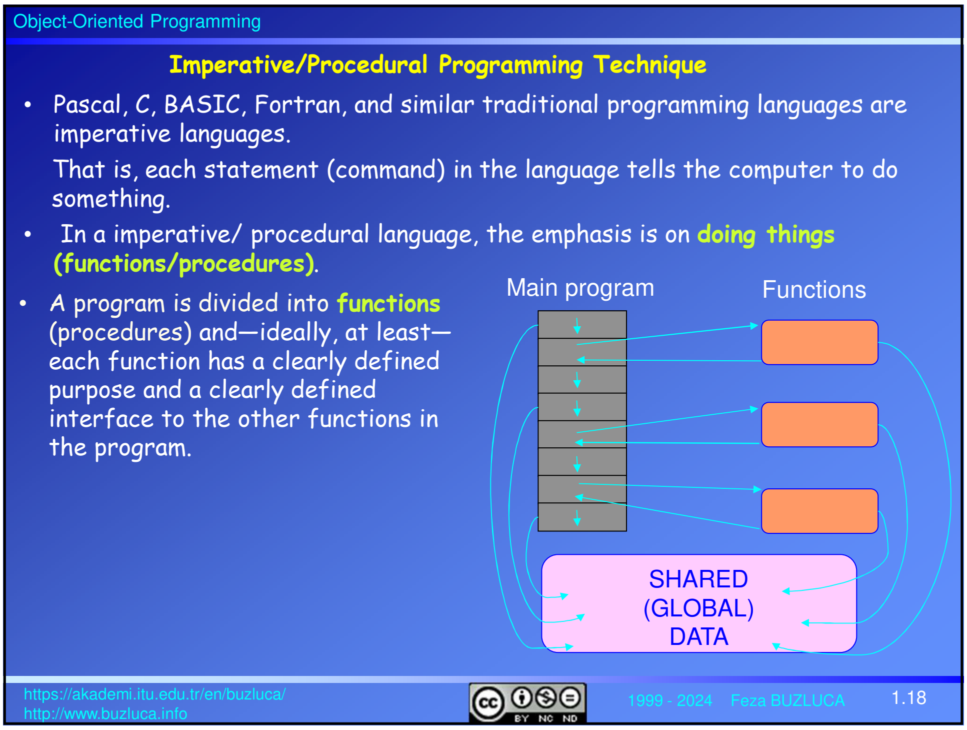
Working drafts of C++23 are also available.

You can get the standard in İTÜ campus from the website of the British Standards Online: http://bsol.bsigroup.com/

Information about C++ standards: https://isocpp.org/std/the-standard

Be aware of programming standards and use compilers that support the current one.

For example, you can use GCC, Clang, or Visual Studio.



Imperative/Procedural Programming Technique

- Pascal, C, BASIC, Fortran, and similar traditional programming languages are imperative languages.

  That is, each statement (command) in the language tells the computer to do something.
- In a imperative/ procedural language, the emphasis is on **doing things (functions/procedures)**.
- A program is divided into **functions** (procedures) and—ideally, at least—each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.

Problems with the Imperative/Procedural Programming

- Procedural programs (functions and data structures) do not model the real world very well.

  The real world does not consist of only functions. The real world consists of objects.
- Data is undervalued; emphasis is on functions.

  Data is, after all, the reason for a program's existence.

  The essential parts of a program about a school, for example, are not functions that display the data or functions that check for correct input, etc.

  Essential parts are student, teacher, and course data.
- Data items and related functions are scattered around the program (not in the same module as objects).
- Global data can be corrupted by functions that have no business changing it.
- Creating new (user-defined) data types is complex.

Imperative programming also has some advantages, and it is also possible to write good programs using procedural programming (e.g., C programs).

However, object-oriented programming offers programmers many advantages to enable them to write high-quality programs.

The Object-Oriented Approach

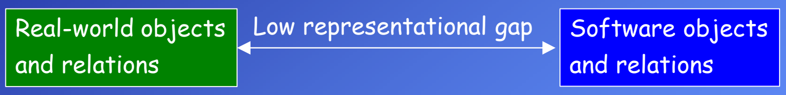
The fundamental principle of object-oriented programming is the "**low representational gap.**"

The real world (problem) consists of objects.

The software system (solution) also consists of objects.

Computer programs may contain computer-world representations of the things (objects) that constitute the solutions to real-world problems.

The close match between objects in the programming sense and objects in the real world increases the quality of the design.


Real-world objects and relations
Low representational gap
Software objects and relations

What kinds of things become objects in object-oriented programs?

- Human entities: Employees, customers, salespeople, workers, manager
- Graphics program: Point, line, square, circle, ...
- Mathematics: Complex numbers, matrix
- Computer user environment: Windows, menus, buttons
- Data-storage constructs: Customized arrays, stacks, linked lists

Example:

If you look at a university system, there are many functions and a lot of complexity.

- Students have IDs, they attend courses, they take grades, and their GPAs are calculated.
- Instructors give courses, perform some industrial and scientific projects, have administrative duties, and their salaries are calculated each month.
- Courses are offered in specific time slots in a classroom. They have a plan.

Considered this way, looking at every element at once and focusing on functions, a university system becomes very complex.

Object-oriented modeling:

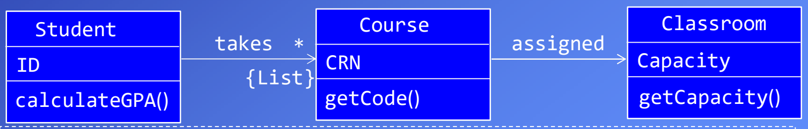
If you wrap what you see in the problem up into **objects**, the system is easier to understand and handle.

- There are students, instructors, courses, and classrooms.
- These objects have behaviors, abilities, or responsibilities.
- There are relations between them.

Thinking in terms of objects:

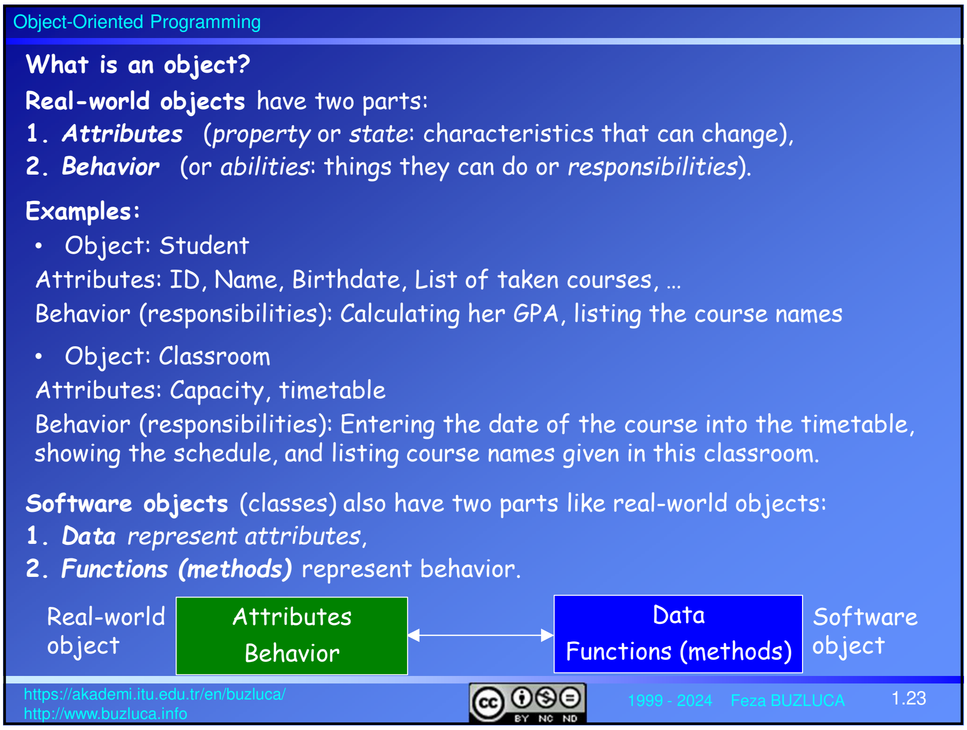
To solve a problem in an object-oriented language, the programmer should consider three factors:

1. What are the **objects** that make up the problem domain?
   Student, course, instructor, classroom, etc.
2. What are the **responsibilities** of objects?
   Students can calculate their GPAs; instructors can enter grades; classrooms can express their capacities, etc.
3. What are the **relations** between objects?
   Students take courses; students contain a list of courses; a master's student is a special type of student, and courses are given in classrooms.



(The Unified Modeling Language (**UML**) is a useful tool to express the model.)

Internal mechanisms and parts that work together are wrapped into a *class*.



What is an object?

Real-world objects have two parts:

1. *Attributes* (*property* or *state*: characteristics that can change),

2. *Behavior* (or *abilities*: things they can do or *responsibilities*).

Examples:

- Object: Student

Attributes: ID, Name, Birthdate, List of taken courses, …

Behavior (responsibilities): Calculating her GPA, listing the course names

- Object: Classroom

Attributes: Capacity, timetable

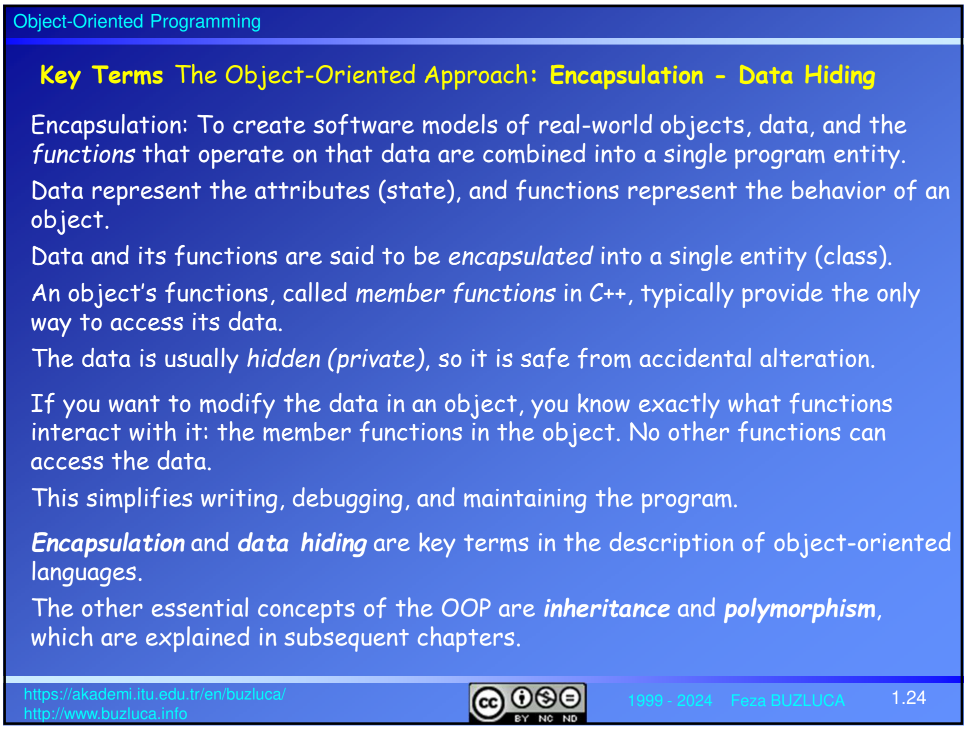
Behavior (responsibilities): Entering the date of the course into the timetable, showing the schedule, and listing course names given in this classroom.

Software objects (classes) also have two parts like real-world objects:

1. *Data* *represent attributes*,

2. *Functions (methods)* represent behavior.

Real-world object	Attributes Behavior	↔	Data Functions (methods)	Software object



Key Terms The Object-Oriented Approach: **Encapsulation - Data Hiding**

Encapsulation: To create software models of real-world objects, data, and the *functions* that operate on that data are combined into a single program entity.

Data represent the attributes (state), and functions represent the behavior of an object.

Data and its functions are said to be *encapsulated* into a single entity (class).

An object's functions, called *member functions* in C++, typically provide the only way to access its data.

The data is usually *hidden (private)*, so it is safe from accidental alteration.

If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object. No other functions can access the data.

This simplifies writing, debugging, and maintaining the program.

Encapsulation and ***data hiding*** are key terms in the description of object-oriented languages.

The other essential concepts of the OOP are ***inheritance*** and ***polymorphism***, which are explained in subsequent chapters.

Example of an Object: A Point in a graphics program

A point on a plane has two attributes; x-y coordinates.

A point's abilities (behavior, responsibilities) are moving on the plane, appearing on the screen, and disappearing.

These responsibilities are determined by the requirements of the stakeholders.

We can create a model for two-dimensional points with the following parts:

- Two integer variables (**x** , **y**) to represent x and y coordinates
- A function to move the point: **move** ,
- A function to print the point on the screen: **print** ,
- A function to hide the point: **hide** .

Model (class) of a point:

UML

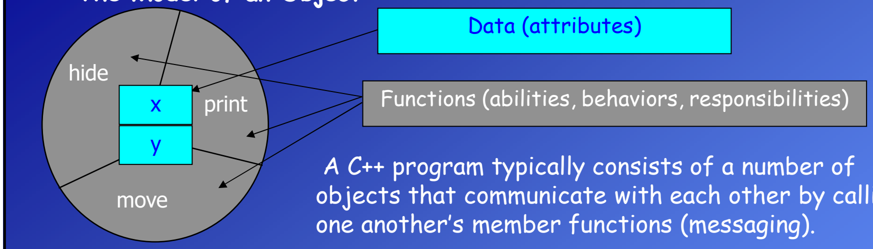
Point
- x, y: Integer
+ void move(int, int) + void print() + void hide()

Once the model (class) of the point has been built and tested, it is possible to create and activate many point objects from this model.

In the example on the right, point1, point2, and point3 are three different objects of the same class (model) Point.

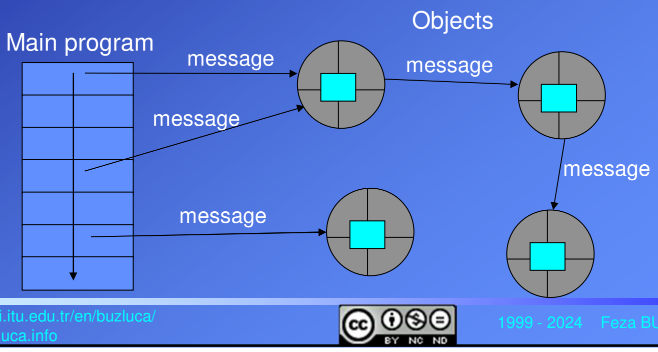
```
Point point1, point2, point3;
     :
point1.move(50,30);
point1.print();
point2.move(0,100);
```

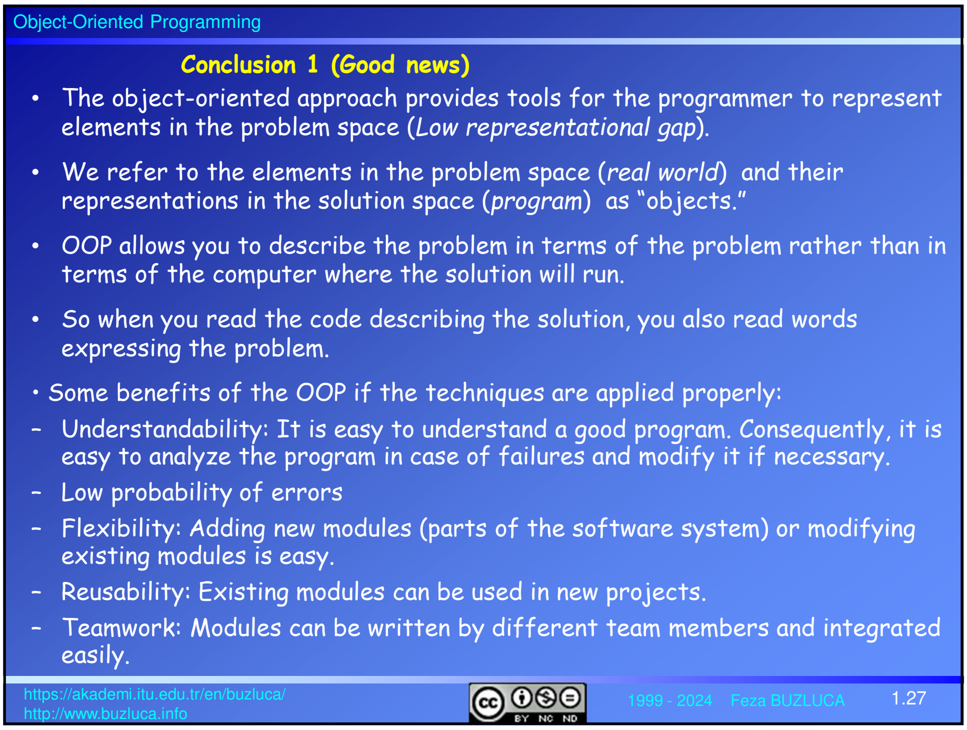
The Model of an Object



A C++ program typically consists of a number of objects that communicate with each other by calling one another's member functions (messaging).

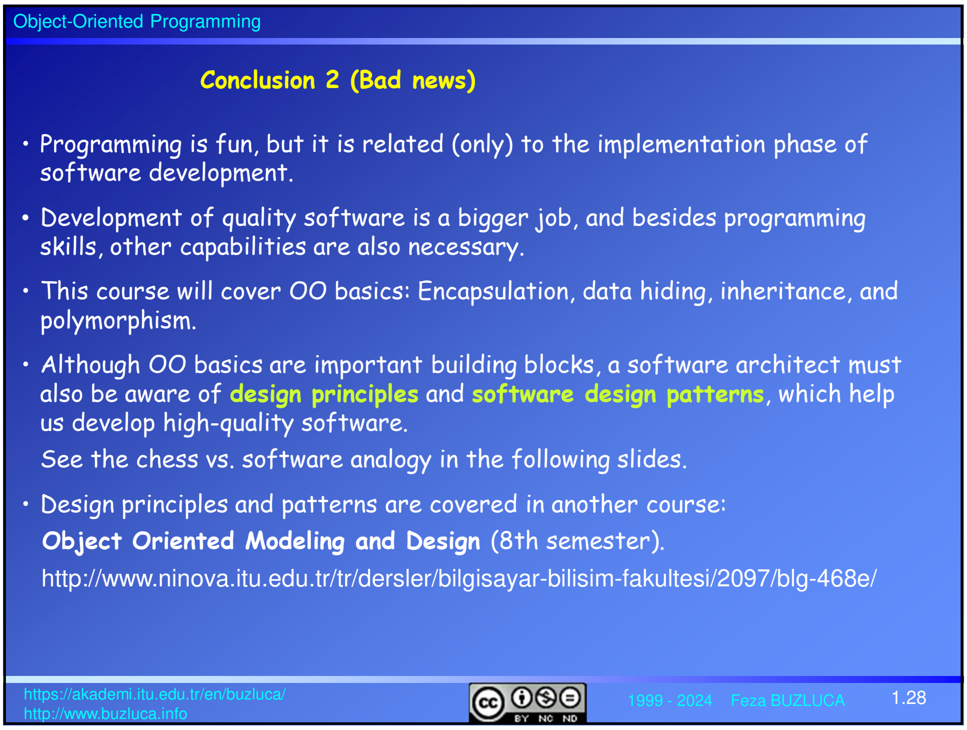
Structure of an object-oriented program in C++:





Conclusion 1 (Good news)

- The object-oriented approach provides tools for the programmer to represent elements in the problem space (*Low representational gap*).
- We refer to the elements in the problem space (*real world*) and their representations in the solution space (*program*) as "objects."
- OOP allows you to describe the problem in terms of the problem rather than in terms of the computer where the solution will run.
- So when you read the code describing the solution, you also read words expressing the problem.
- Some benefits of the OOP if the techniques are applied properly:
  - Understandability: It is easy to understand a good program. Consequently, it is easy to analyze the program in case of failures and modify it if necessary.
  - Low probability of errors
  - Flexibility: Adding new modules (parts of the software system) or modifying existing modules is easy.
  - Reusability: Existing modules can be used in new projects.
  - Teamwork: Modules can be written by different team members and integrated easily.



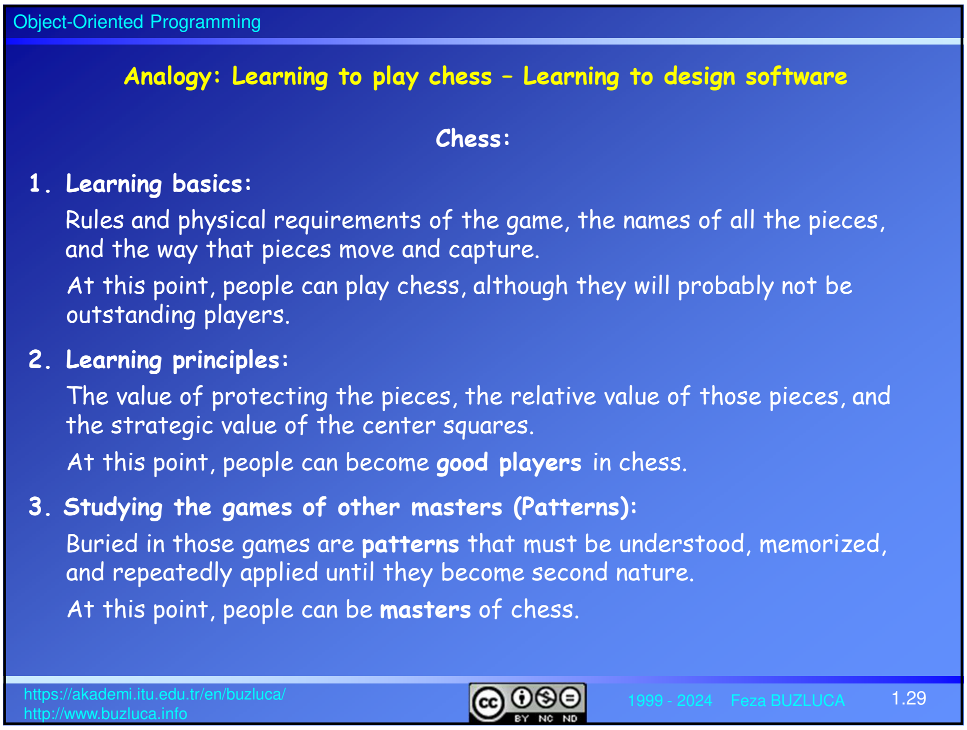
Conclusion 2 (Bad news)

- Programming is fun, but it is related (only) to the implementation phase of software development.
- Development of quality software is a bigger job, and besides programming skills, other capabilities are also necessary.
- This course will cover OO basics: Encapsulation, data hiding, inheritance, and polymorphism.
- Although OO basics are important building blocks, a software architect must also be aware of **design principles** and **software design patterns**, which help us develop high-quality software.

  See the chess vs. software analogy in the following slides.
- Design principles and patterns are covered in another course:

  Object Oriented Modeling and Design (8th semester).

  http://www.ninova.itu.edu.tr/tr/dersler/bilgisayar-bilisim-fakultesi/2097/blg-468e/



Analogy: Learning to play chess – Learning to design software

Chess:

1. **Learning basics:**

   Rules and physical requirements of the game, the names of all the pieces, and the way that pieces move and capture.

   At this point, people can play chess, although they will probably not be outstanding players.

2. **Learning principles:**

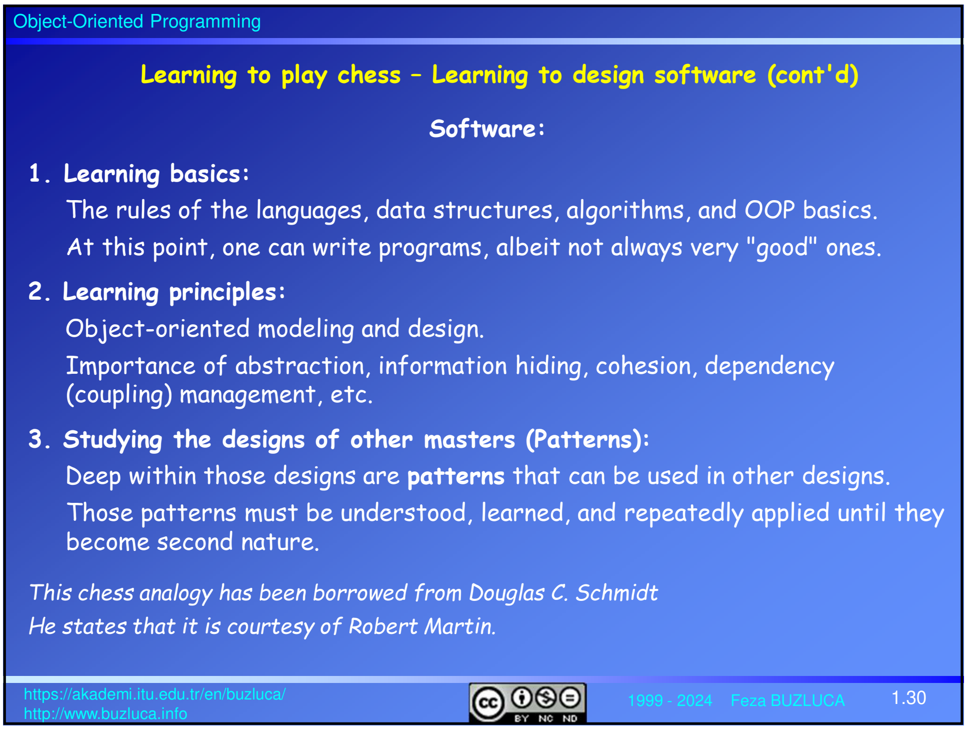
   The value of protecting the pieces, the relative value of those pieces, and the strategic value of the center squares.

   At this point, people can become **good players** in chess.

3. **Studying the games of other masters (Patterns):**

   Buried in those games are **patterns** that must be understood, memorized, and repeatedly applied until they become second nature.

   At this point, people can be **masters** of chess.



Learning to play chess – Learning to design software (cont'd)

Software:

1. **Learning basics:**

   The rules of the languages, data structures, algorithms, and OOP basics.

   At this point, one can write programs, albeit not always very "good" ones.

2. **Learning principles:**

   Object-oriented modeling and design.

   Importance of abstraction, information hiding, cohesion, dependency (coupling) management, etc.

3. **Studying the designs of other masters (Patterns):**

   Deep within those designs are **patterns** that can be used in other designs.

   Those patterns must be understood, learned, and repeatedly applied until they become second nature.

This chess analogy has been borrowed from Douglas C. Schmidt

He states that it is courtesy of Robert Martin.