University of Sarajevo
Faculty of Electrical Engineering
Department of Computer Science and Informatics

NatId Documentation

Artificial intelligence

Authors: Benjamin Bandić, Nedim Bečić
Professor: Izudin Džafić
Date: 18.10.2025

Sadržaj

1 C++, IDEs, CMAKE, Dynamic libraries

1.1 History of C++

The development of the C++ programming language began with Bjarne Stroustrup, a Danish computer scientist, at Bell Laboratories. His work commenced during his doctoral thesis, driven by a vision to enhance the C language by incorporating object-oriented features.

Key Milestones in C++ Development:

- **1979.** - The first implementation of the language, then known as "C with Classes," marked the genesis of what would evolve into C++.
- **1982.** - The initial "Reference manual" was published, offering the first official insight into the language's functionalities.
- **1984.** - The language was officially renamed C++.
- **1985.** - The release of the first commercial version, C++ 1.0, facilitated wider adoption of the language.
- **1989.** - This was followed by the second commercial version, C++ 2.0, which introduced further enhancements.
- **1998.** - C++ 98 became the first ISO standard version, initiating a period where all subsequent versions would be certified according to ISO standards.
- **2003.** - C++03 was released, primarily focusing on correcting errors present in C++98.
- **2009.** - The C++0x version was introduced, serving as a preliminary iteration for upcoming major releases.
- **2011.** - With the C++11 version, significant changes and new features were introduced, modernizing the language considerably.
- **2014.** - C++14 brought improvements and refinements to the C++11 version.
- **2017.** - C++17 encompassed almost all necessary tools for multi-platform application development, including support for file systems, threads, and functional programming. At this stage, standards for network (internet) communication and graphical user interfaces (GUIs) were still lacking.
- **2020.** - C++20 continued the trend of language expansion, adding functionalities such as a calendar and timezone library, import statements, and modules, further solidifying C++'s position as a comprehensive language for diverse applications.

1.2 Advantages of C++

C++ offers a multitude of benefits that make it a compelling choice for various development scenarios:

- **Portability** - The use of standard C++ versions guarantees that applications will compile on all operating systems equipped with a C++ compiler.

- **Wide Range of Applications** - C++ is uniquely positioned to cover a vast spectrum of development areas, including:
  - Driver development (low-level systems)
  - Embedded systems
  - Operating systems
  - 3D animations and games
  - Desktop applications
  - Database applications

  It is arguably one of the only languages that covers such a broad range of domains.

- **Object-Oriented** - C++ fully supports Object-Oriented Programming (OOP) concepts such as polymorphism, encapsulation, inheritance, and abstraction. These principles facilitate cleaner, more manageable code and offer significant advantages over procedural programming languages.

- **Fast Execution with Low Memory Requirements** - C++ applications are known for their high performance and efficient use of system resources, making them ideal for performance-critical tasks.

- **Similarity to Other Programming Languages** - Its syntax and paradigms share similarities with languages like C, C#, and Java, making it relatively easier for developers familiar with these languages to learn C++.

- **Large Community of C++ Users** - A vast and active community supports C++ development. Considering that C is a subset of C++, C++ (including its C heritage) could be considered the most frequently used programming language globally.

- **Significant Market Opportunities** - Proficiency in C++ opens doors to numerous career opportunities across various industries due to its critical role in high-performance computing, systems programming, and game development, among others.

1.3 Disadvantages of C++

While powerful, C++ comes with certain drawbacks that developers must consider:

- **Great Power Comes with Great Responsibility** To enable "low-level" implementations such as drivers, memory management is left to the C++ programmer, primarily through the use of pointers.

- **Lack of Inherent Safety** This direct memory management can lead to safety issues, most notably:
  - **Buffer Overflows:** A critical vulnerability where code can write data outside the boundaries of dynamically allocated memory.
  - **Critical Attention to Memory Bounds:** It is crucial to ensure that code never, under any circumstances, allows data to be written beyond the allocated memory space.
  - **Common Attack Vector:** Buffer overflows are one of the most frequently exploited attack vectors on systems. If an attacked application possesses administrator privileges, an attacker can manipulate data on the computer (send it anywhere, modify, delete, etc.).

Despite these potential issues, this problem is "curable" through careful programming practices, which is why C++ remains one of the most widely used languages.

1.4 A Short History of Personal Computers and Operating Systems

The evolution of personal computers and their operating systems has been a journey of constant innovation, shaping the digital landscape we know today.

- **Mid-1960s:** The inception of Unix at Bell Laboratories, initially utilized on large "main-frame" systems, laid foundational concepts for future operating systems.
- **1978:** Apple DOS emerged as an early operating system for personal computers.
- **1981:** IBM introduced PC-DOS, quickly followed by Microsoft's widely adopted MS-DOS, establishing a dominant standard for early personal computing.
- **January 1984:** MacOS (v 1.0) revolutionized user interaction by becoming the first widely available system with a Graphical User Interface (GUI) and introducing the mouse as a primary input device.
- **1985:** MacOS (v 2.0) further enhanced its capabilities by adding support for laser printers, cementing an architecture for personal computers that largely influences design even today.
- **1985:** Microsoft developed Windows 1.0 as an extension for DOS.
  - The underlying operating system remained DOS.
  - This extension was generally considered difficult to use until version 3.0.
- **1988:** MacOS (v 6.0) continued Apple's refinement of its GUI-based system.
- **1990:** Windows 3.0, still a DOS extension, gained significant traction in the market.
- **1991:** Linux 0.01-0.1 marked the beginning of the open-source operating system movement.
- **1992:** Windows 3.1 brought further improvements and stability to Microsoft's graphical environment.
- **1993:** Free BSD, an open-source Unix-like operating system, was released, serving as a base for many other systems, including Apple's operating systems and PlayStation 4.
- **1994:** Red Hat Linux appeared, leading to a proliferation of various Linux distributions such as Ubuntu, SUSE, and Fedora.
- **1995:** Windows 95 represented a significant turning point for Microsoft, being its first 32-bit operating system and introducing a modern user interface.
- **2001:** MacOS X was released, initiating an annual update cycle that continued until the advent of macOS 11 in November 2020.
- **2007:** iPhone OS 1.0 (later renamed iOS 1.0) became a monumental breakthrough in mobile software development, defining the smartphone experience.
- **2008:** Android 1.0, a Linux-based OS with a Java GUI, was introduced.
  - Initially, it was considerably slower and offered significantly poorer performance compared to iPhone OS.
- **2015:** Windows 10 was launched, aiming to unify the user experience across various devices.
- **2020:** macOS 11 (Big Sur) marked a new era for Apple's desktop operating system.

  - It introduced the transition to ARM-based processors, signaling a break from Intel.
  - Following this, no new Mac computers with x86-based processors would be produced.

Operating systems from application development perspective

Developing applications across different operating systems presents unique challenges and opportunities, particularly concerning Graphical User Interfaces (GUIs).

- **C++ for Non-GUI Portable Applications** Utilizing C++ alongside system-level OS calls allows for straightforward and rapid development of portable applications that do not require a GUI.
- **GUI Application Development Across Modern OS** GUI application development showcases the greatest diversity among modern operating systems.
- **macOS: The Most Consistent Approach**
  - The **Cocoa framework** provides a highly consistent approach to developing GUI applications for macOS, from version X to the present day.
  - Development languages used include **Objective-C (2.0)** and, more recently, **Swift**.
- **Windows: The Most Inconsistent Approach** Windows has historically had the most varied and fragmented approach to GUI development.
  - Initially, the **Win32 API** was developed (primarily for C, with an MFC extension for C++). This API is still in use today, though applications built with it may not share the modern aesthetic of newer applications.
  - **Windows Forms**, **Windows Presentation Foundation (WPF)**, and **Universal Windows Platform (UWP)** are platforms for application development primarily using the C# language.
  - **WinUI 1.0, 2.0**: These platforms were developed using C# and a non-standard version of C++ (C++/CX).
  - **WinUI 3.0**: This platform enables development using C# and standard C++.
  - The **Windows App SDK** unifies Win32 and WinUI 3.0, allowing for the modernization of Win32 applications (current version 1.1 is not yet extensively used for internal Microsoft development).
- **Linux GUI Development**
  - Linux GUIs were traditionally often based on the **Xlib library**.
  - Modern Linux desktops are predominantly built on the **GTK library** (used by the Gnome Desktop environment) and the **Qt library** (used by the KDE desktop environment).
  - Both GTK and Qt also facilitate the development of GUI applications on macOS and Windows systems, offering cross-platform capabilities.

1.5 Tools for Software Development with C++

The development of C++ applications relies on a suite of tools that have evolved from basic command-line utilities to sophisticated Integrated Development Environments(IDEs).

- **Editor, Compiler, Linker, Debugger**

  - Initially, applications were written using simple text editors.
  - The written code was functionally grouped via ‘make‘ scripts.
  - A **compiler** was used to convert the source code into object code.
  - A **linker** was then used to generate the executable code for the target OS.
  - In case of problems, a **debugger** was invoked from the console to inspect variable values by setting breakpoints.
  - **Commonly used compilers** include ‘clang‘ (macOS), ‘cl‘ (Windows), and ‘gcc‘ (Linux).
- **Most Commonly Used Environments**
  - **Xcode** is the most frequently used environment for macOS application development.
    * It is exclusively available on macOS.
    * Offers exceptional integration with the OS.
    * Provides significantly faster compilation compared to other tools.
    * It is a free tool for macOS.
    * CMake can generate projects for Xcode.
  - **Visual Studio (2022)**:
    * The Community edition is free.
    * Professional and Enterprise versions are also available.
    * Exclusively available on Windows OS.
    * Considered one of the most widely used IDEs overall.
    * Features integration with CMake.
  - **Visual Studio Code**:
    * Available on Windows, Linux, and macOS.
  - **Qt Creator**: Primarily used for Qt application development.
    * Known for its portability.
    * Available for most operating systems.
  - Other environments include Code Blocks, Eclipse, etc.

Software Engineering Principles

Complex applications are typically divided into smaller parts for two main reasons: to enable code reuse through (dynamic) libraries, and to allow different teams to work independently on their respective modules. Interaction between these libraries is managed via defined interfaces. Modules can contain one or more static or dynamic libraries.

The distinction between static and dynamic libraries is crucial. A static library must be linked directly into another dynamic library or an executable application. If an error exists in a static library, it necessitates recompiling not only that library but also all other components (dynamic libraries and applications) that use it. In contrast, if an error is found in a dynamic library, only that specific library

needs to be recompiled, relinked, and overwritten. This ability to update components independently is the primary advantage of dynamic libraries over static ones.

Dynamic libraries contain a list of variables, methods, and classes that they export and make available to other parts of an application. These libraries can be loaded automatically at runtime or on user demand (e.g., using ‘LoadLibrary‘ on Windows). The on-demand loading capability, since they don't need to be statically linked, enables the implementation of plug-in components. If a library provides a certain function that satisfies a defined interface, it can be added to a list of available plug-ins. Due to these advantages, working without dynamic libraries is inconceivable in modern software engineering. Standard extensions for dynamic libraries are ‘.dylib‘ on macOS, ‘.dll‘ on Windows, and ‘.so‘ on Linux.

Integrated Development Environments (IDEs)

Initially, developing applications involved a complex and slow process of using text editors in conjunction with separate command-line compilers, linkers, and debuggers. The solution to this inefficiency was the integration of these tools into a single, unified application for software development, known as an Integrated Development Environment (IDE). IDEs have become the modern standard for application development, significantly streamlining the entire workflow by providing a comprehensive suite of tools in one interface.

1.6 Creating Portable Applications - CMake

Development of Complex Portable Applications

Efficient development of portable applications mandates careful consideration of the chosen Integrated Development Environment (IDE). One approach to achieving portability is by using IDEs available across all operating systems.

However, this method comes with at least two key problems:

- IDEs that cover most operating systems often lack good integration with the native debugger of the system they are running on.
- If a developer is not accustomed to that system, they will face a very difficult adaptation period to the new IDE.

What is CMake and what is its purpose?

CMake is a cross-platform project configuration system that operates independently of compilers, linkers, IDEs, and the operating system itself. CMake can generate compiler-linker scripts for console tools as well as project files for most existing IDEs. This approach enables universal and straightforward project configuration, allowing the user to convert it into their preferred IDE based on their platform (OS) and preferences. This method achieves both portability and efficiency in developing portable applications.

Primarily, CMake is designed for configuring projects that use C and C++ languages (Objective-C is also supported), which is why it was named CMake. Therefore, CMake does not have its own compiler, linker, debugger, or editor. It is a scripting framework for configuring and converting project configurations, thereby enabling efficient portability in project development.

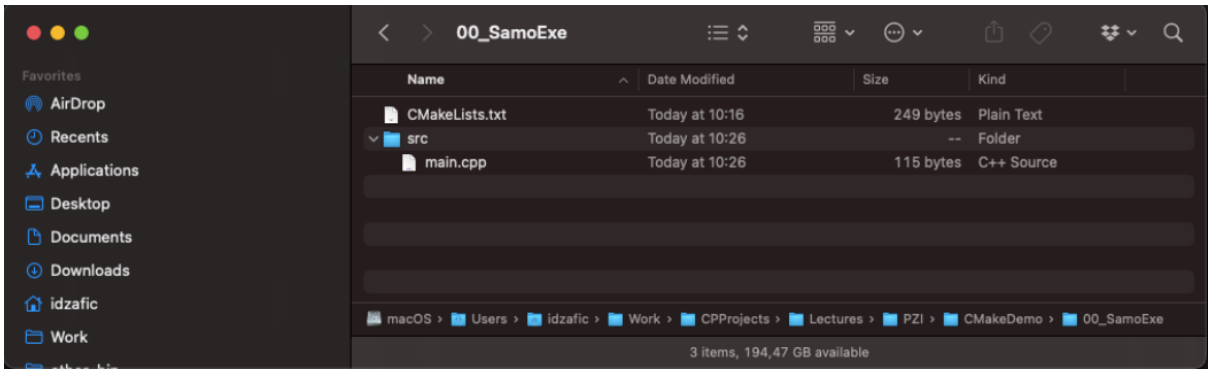
Installation and usage

CMake is free and available for download from: *https://cmake.org/download/*. After installation, CMake

requires a ‘CMakeLists.txt‘ file in one of the project’s directories (usually in the project root). In addition to ‘CMakeLists.txt‘, one or more ‘.cmake‘ files can optionally be used and included in ‘CMakeLists.txt‘.

For easier understanding, the principle of configuring CMake projects will be explained using a simple application without additional libraries. This will be followed by an explanation of how to add dynamic libraries. Finally, a sample CMake project will be created that only needs to be copied to another location, with project names changed as needed. For simplicity and uniformity, it is suggested to use a **‘src‘** directory for grouping source (‘.cpp‘) and header (‘.h‘) files within each project.

The example structure for a basic application ("Hello from CMake") is shown in the image below:



Slika 1: Example structure for a basic CMake application.

Thus, the minimum requirement is the existence of ‘CMakeLists.txt‘ which defines the project. The first line should contain the minimum required CMake version. For example, for the required version 3.18, you would have:

```
cmake_minimum_required(VERSION 3.18)
```

Slika 2: Example structure for a basic CMake application.

After that, the project name needs to be defined. Since the project name can appear in multiple places in the script, it is convenient to define a new CMake variable using ‘set‘ and then use it by referencing the variable name, e.g.,

```
set(PROJECT_NAME samoExe)    #Unijeti naziv solution-a
project(${PROJECT_NAME})
```

Slika 3: Example structure for a basic CMake application.

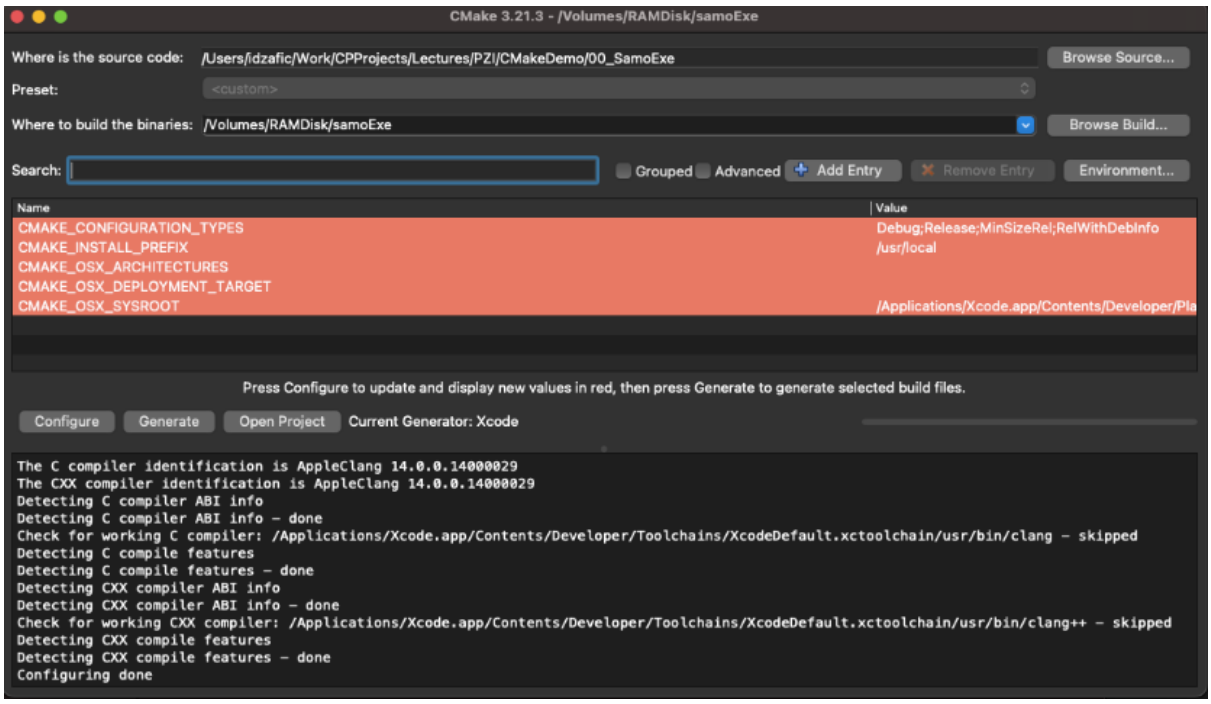
Creating IDE Projects from CMake

Visual conversion requires starting the CMake graphical interface. After selecting the directory containing `'CMakeLists.txt'` and choosing the location where the IDE project file will be generated, then clicking on `'Configure'` (and selecting the IDE) and `'Generate'`, the configuration file is created.

Note: The generated IDE configuration file should always be outside the source code workspace. In our case, if the root directory for the workspace is `'Work'`, then locations within `'Work'` should not be selected as destinations for IDE projects.

Generating a Project for an IDE For this example:

- The selected location for `CMakeLists.txt` is: `/Users/idzafic/Work/CPProjects/Lectures/PZI/CMakeDemo/00_SamoExe`.
- The selected destination for the IDE project is `/Volumes/RAMDisk/samoExe`.
- After clicking the `Configure` button, confirming the creation of a new directory, selecting Xcode as the target IDE, and clicking the `Generate` button, the appearance of the CMake application is as shown below:



Slika 4: CMake GUI after configuring and generating an Xcode project.

Following these steps, the project is ready for work in the Xcode IDE (e.g., "Demo").

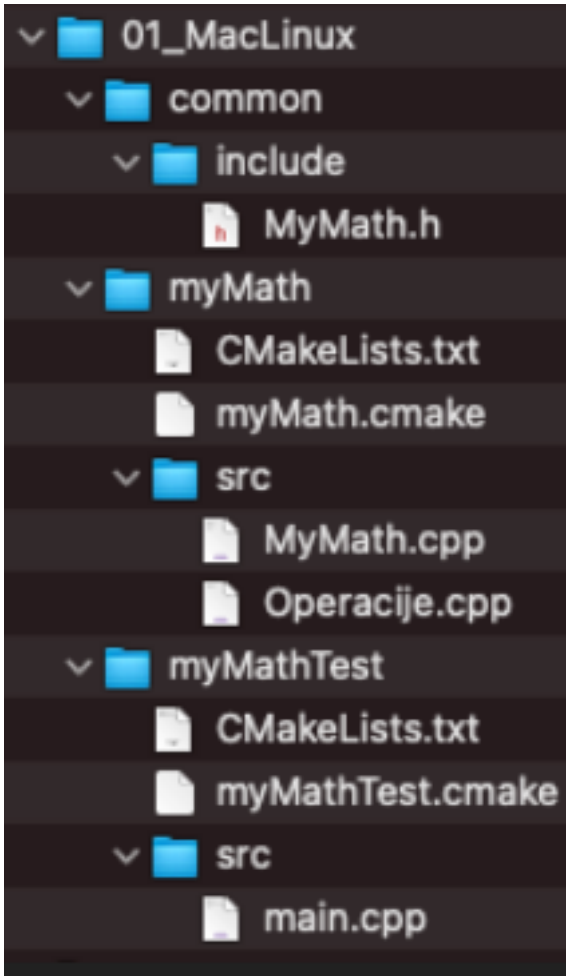
1.7 Dynamic Libraries

Dynamic libraries can either be created as part of your project or obtained as pre-compiled (external) binaries. In both scenarios, it is essential to know the location of their interface files (typically `.h` header files).

Creating Internal Dynamic Libraries

If you are developing a dynamic library within your own project, it is straightforward to add. This is done by creating a separate directory for its code and a corresponding CMake (`.cmake`) file that describes it.

To illustrate, our starting application will be extended with a dynamic library that exports simple mathematical operations, with the structure shown in the image below (on the right). The `myMathTest` application (executable) intends to use specific methods provided by the `myMath` dynamic library.



Slika 5: Extended application structure with the `myMath` dynamic library.

The dynamic library (`myMath` module) is defined within `myMath.cmake` using CMake's `add_library` command:

- The first line defines a CMake variable with the library's name.
- The second line defines the dynamic library with its associated `.cpp` and optional `.h` files. For example:

```
set(MYMATH_SHLIB_NAME myMath)

add_library(${MYMATH_SHLIB_NAME} SHARED ${CMAKE_CURRENT_LIST_DIR}/src/Operacije.cpp
                                        ${CMAKE_CURRENT_LIST_DIR}/src/MyMath.cpp
                                        ${CMAKE_CURRENT_LIST_DIR}/../common/include/MyMath.h)
```

Slika 6: Example of adding a library

- Omitting the `SHARED` keyword would create a static library instead.

After defining the library, it needs to be included in the executable application (`myMathTest`).

Modifying the Executable Application

To include your own dynamic library, you must modify the `CMakeLists.txt` of the executable application. For our example, the `CMakeLists.txt` located in the `myMathTest` folder is modified to include `myMath.cmake`:

```
include(../myMath/myMath.cmake)
```

Following this, it is necessary to declare the executable application's dependency on the dynamic library using `add_dependencies`:

```
add_dependencies(executableModuleName libraryModuleName)
```

This command guarantees two things:

- The dynamic library required by the executable application will be compiled and linked first.
- The library containing the exported symbols will be included in the linking process of the executable application.

The complete `CMakeLists.txt` file for our example, which includes its own dynamic library, would be:

```
cmake_minimum_required(VERSION 3.18)

set(SOLUTION_NAME myMathTestMacLin) #Unijeti naziv solution-a

project(${SOLUTION_NAME})

#ukljuci prvi subproject - generisanje shared libraray (dll)
include(../myMath/myMath.cmake)

#ukljuci drugi subroject - executable koji korisi myMath shared library
include(myMathTest.cmake)

#dodaj ovisnost (ona garantuje da se prvo kompajlira shared library pa onda executable)
add_dependencies(${MYMATH_SHLIB_TEST_NAME} ${MYMATH_SHLIB_NAME})
```

Slika 7: Complete `CMakeLists.txt` for our example

1.8 Compiler Dependency and symbol export

Up to this point, we haven't specified which symbols are exported from our dynamic library. This is because certain compilers (like `gcc`, `clang`, etc.) by default export all symbols if nothing explicit is

stated. Other compilers, however, export nothing unless symbols are explicitly specified (e.g., `cl` for Visual Studio); these compilers require explicit declaration of exported symbols.

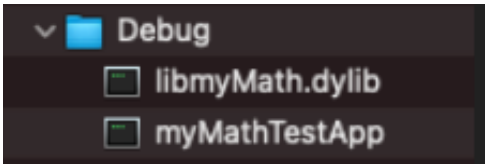
The advantage of the first approach is that the programmer doesn't need to manage the specification of exported symbols from a dynamic library, leading to easier programming. However, this method has drawbacks:

- Slower loading (startup) of the executable application.
- Symbols are resolved dynamically, requiring the system to find specific methods among many that might not even be used.
- Potential for "collision" of exported symbols.

To understand the difference, let's examine what the `clang` compiler (Xcode on macOS) exports versus what the Visual Studio compiler (on Windows OS) exports.

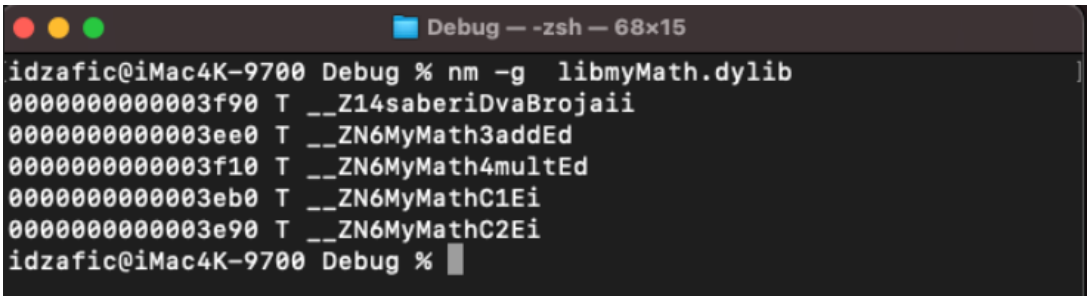
Compiler Dependency - macOS and Linux

Xcode (macOS) uses the `.dylib` extension for dynamic libraries. For our example, compiling in debug mode generates two files:



Slika 8: Generated files in debug mode on macOS.

If we list the exported symbols in the macOS terminal (using the command `nm -g libraryName.dylib`):



Slika 9: Output of `nm -g` command on macOS.

As can be seen, all symbols defined within the dynamic library are exported:

- Function names.
- Class and method names.
- All global variables used (not present in this example).

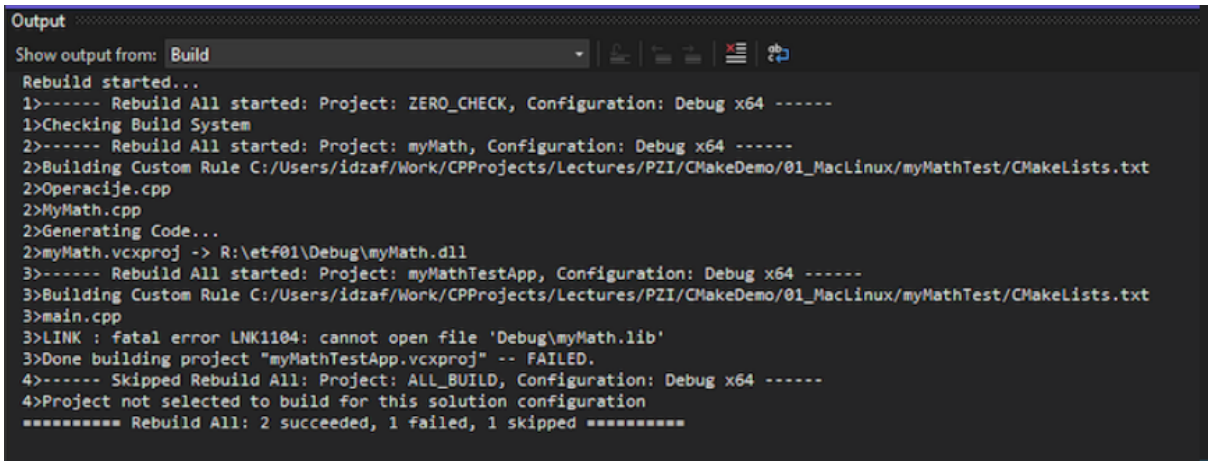
A similar result would be obtained on Linux OS (using `clang`, `gcc`, etc.), where the `.so` extension is used for dynamic libraries instead of `.dylib`.

Compiler Dependency - Windows (Visual Studio)

If such a definition for a dynamic library were used, Visual Studio (VS) would not be able to link the executable application. The reasons are:

- VS generates two files for dynamic libraries: `.lib` and `.dll` extensions.
- The `.dll` file contains the executable code of the dynamic library.
- The `.lib` file contains a list of exported symbols that the `.dll` exports, as well as a link to the `.dll` file.
- The VS compiler always requires the `.lib` file. Therefore, for linking, it is necessary to specify the location of the `.lib` file, and the `.dll` should either be in the local folder or in the system's PATH.

For our example, VS would generate the following output with an error like:
"cannot open file 'Debug\myMath.lib'"



Slika 10: Visual Studio linker error when `.lib` file is missing.

Q) Why did the linking of the executable application fail?

A) Since no symbol to be exported from the dynamic library was specified, the `.lib` file was not created, and therefore could not be found. If we look at the location with the output files for Debug mode, it is clear that the `.dll` file exists (`.pdb` is an auxiliary file for debugging) but the `.lib` file is missing.

Q) What needs to be done to generate the library and successfully link the executable application code?

A) VS (Windows) requires that all symbols to be exported from a library are declared as exported, and all symbols to be imported from another library are declared as imported. Therefore, if we consider a single symbol, it must be declared for export within the dynamic library where it is implemented, and declared for import in all other libraries/executable applications. For this purpose, an additional `.h` file will be created to define a macro that behaves as described above.

Since this introduces Windows-specific features that do not exist on other platforms, it is necessary to ensure that this part of the code is active only on the Windows platform. Thus, for the code to function on all OS, three things need to be known:

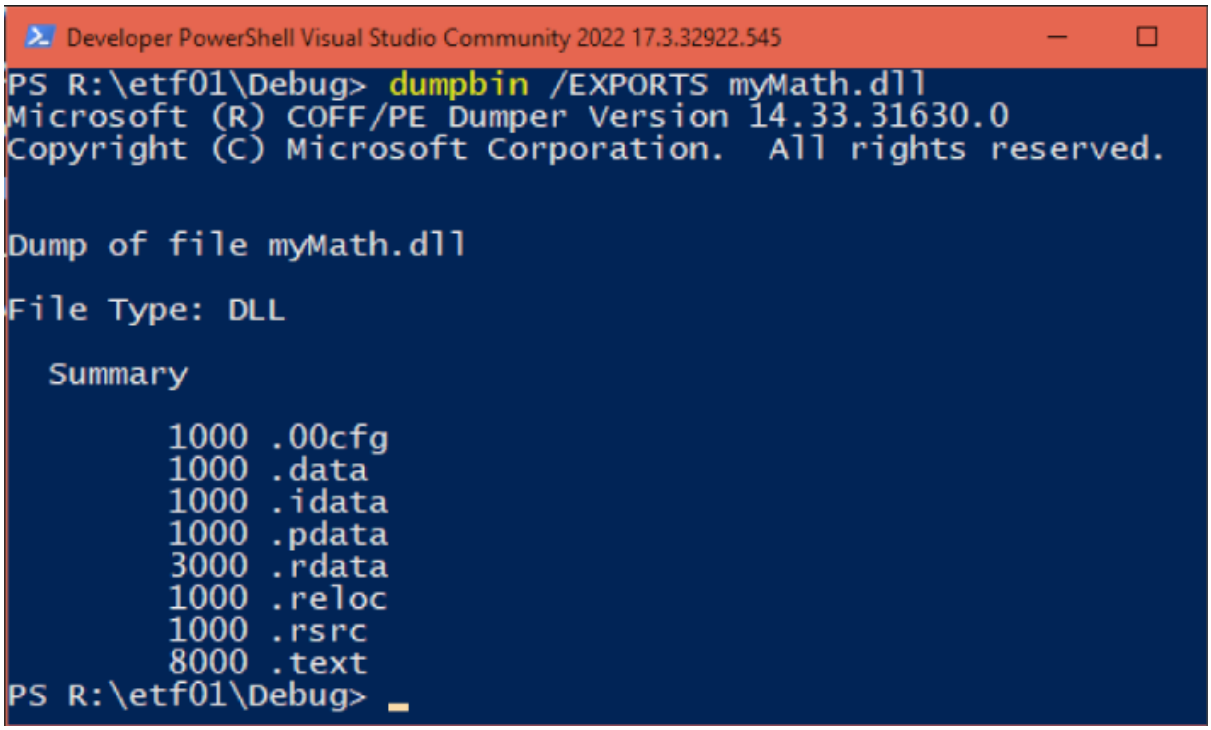
- On which operating system we are compiling (Windows, macOS, Linux).

- Whether symbols are being exported or imported.
- Directives for importing or exporting symbols.

Verifying Windows Export Status

To verify that Visual Studio has not exported any symbols from the `.dll`, you can do the following:

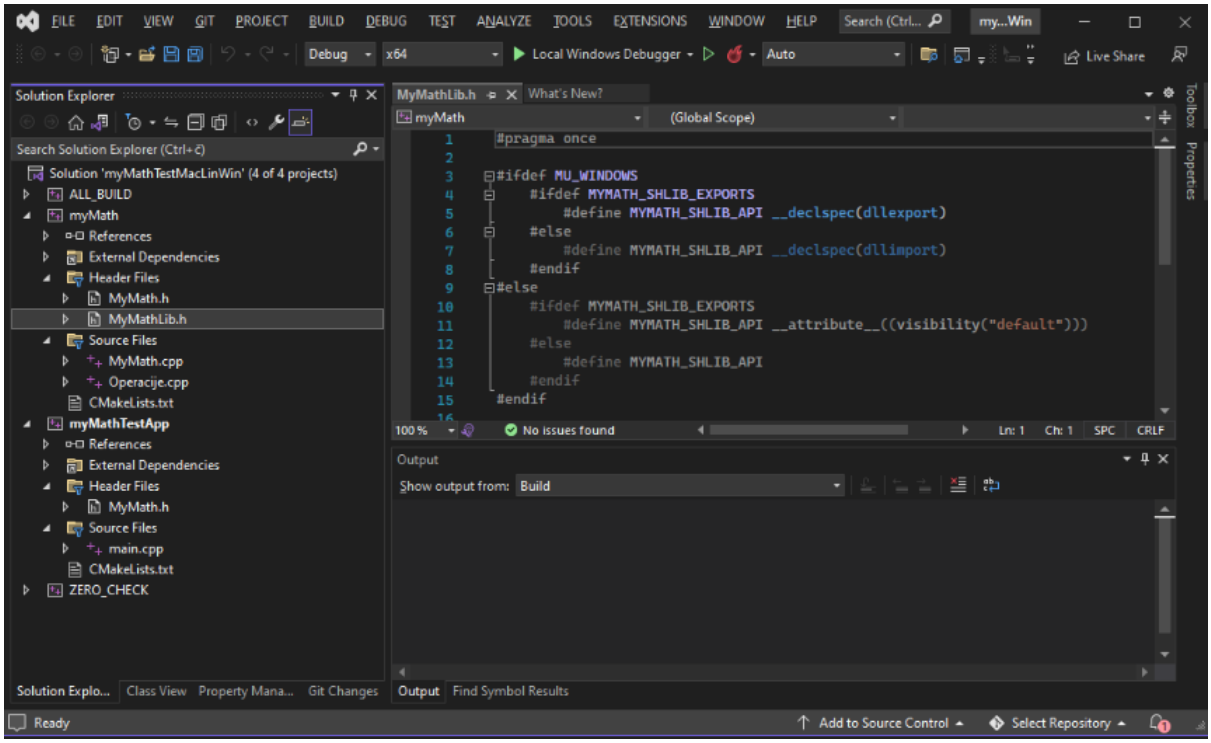
- Start Developer PowerShell (launched from VS, under the Tools submenu).
- Navigate to the folder containing the `.dll`.
- Execute the command `dumpbin /EXPORTS myMath.dll`.



Slika 11: Output of `dumpbin /EXPORTS` showing no exported symbols.

Changes in Source Code Structure

A new header file has been added, and the `.cmake` file has been modified to pass information about the OS and the fact that only the library exports symbols.



Slika 12: Changes added to the source code structure

OS Detection and Symbol Export/Import Directives

For the purpose of detecting the type of operating system for which the source code is compiled, three directives will be used, of which only one can be active (set):

- `MU_MACOS` - active only if OS == macOS
- `MU_WINDOWS` - active only if OS == Windows
- `MU_LINUX` - active only if OS == Linux

CMake allows for simple OS detection and will be used for this purpose (demonstration follows).

The definition of whether our `myMath` library exports symbols or if someone else imports its symbols is defined (in this example) with `MYMATH_SHLIB_EXPORTS`. This pre-compile variable needs to be set only for the `myMath` library. This way, in the case of Windows, it will be declared as:

```
#define MYMATH_SHLIB_API __declspec(dllexport)
```

Which essentially means that all symbols (variables, functions, and classes) marked with `MYMATH_SHLIB_API` will be exported from the `myMath` library. In the case of other systems, the code segment that indicates the default operation (export) applies.

If a new dynamic library is to be created, this template can be applied. It is necessary to have unique identifiers for each new dynamic library:

- Change `MYMATH_SHLIB_EXPORTS` to a new name (no duplication allowed), e.g., `XXXX_XXXX_EXPORTS`.
- Change `MYMATH_SHLIB_API` to a new name (no duplication allowed), e.g., `XXXX_XXXX_API`.

- Everything else remains unchanged.

Marking Symbols for Export

To export something now, it is necessary to modify the code to "mark" the symbols (variables, functions, and classes) that we want to export. For this purpose, we use `MYMATH_SHLIB_API` (or `XXXX_XXXX_API` for another dynamic library).

Since the definitions of what the `myMath` library should export (interface) are defined in `myMath.h` located in the common directory, its content needs to be modified to see the desired definitions in `myMathLib.h` (which is located in the same common folder):

```
1 #pragma once
2 #include "MyMathLib.h"
3
```

Slika 13: Modifying `myMath.h` to include `myMathLib.h` definitions.

The function ‘saberiDvaBroja‘ (add two numbers) is exported by prepending `MYMATH_SHLIB_API` to its definition:

```
3
4 MYMATH_SHLIB_API int saberiDvaBroja(int a, int b);
```

Slika 14: Example of exporting a function using `MYMATH_SHLIB_API`.

Classes are exported by placing `MYMATH_SHLIB_API` between the ‘class‘ keyword and the class name:

```
5
6  class MYMATH_SHLIB_API MyMath
7  {
8  protected:
9      double _currVal = 0;
10 public:
11     MyMath(int initialVal);
12     double add(double toAdd);
13     double mult(double toMult);
14 };
```

Slika 15: Example of exporting a class using `MYMATH_SHLIB_API`.

Modifying `myMath.cmake` for OS Detection and Directives

To successfully generate the project, the `myMath.cmake` file needs to be modified. In it, OS detection is performed, and the `MYMATH_SHLIB_EXPORTS` directive is defined. It is crucial that no other `.cmake` file (project) has the pre-compiler variable `MYMATH_SHLIB_EXPORTS` defined. This ensures that only ‘myMath‘ exports symbols, and others import them.

```
set(MYMATH_SHLIB_NAME myMath)

add_library(${MYMATH_SHLIB_NAME} SHARED ${CMAKE_CURRENT_LIST_DIR}/src/Operacije.cpp
                                        ${CMAKE_CURRENT_LIST_DIR}/src/MyMath.cpp
                                        ${CMAKE_CURRENT_LIST_DIR}/../common/include/MyMath.h
                                        ${CMAKE_CURRENT_LIST_DIR}/../common/include/MyMathLib.h)

#CMake detektuje OS
if (WIN32)
    target_compile_definitions(${MYMATH_SHLIB_NAME} PUBLIC MYMATH_SHLIB_EXPORTS MU_WINDOWS)
elseif(APPLE)
    target_compile_definitions(${MYMATH_SHLIB_NAME} PUBLIC MYMATH_SHLIB_EXPORTS MU_MACOS)
else()
    target_compile_definitions(${MYMATH_SHLIB_NAME} PUBLIC MYMATH_SHLIB_EXPORTS MU_LINUX)
endif()
```

Slika 16: New modifications to `myMath.cmake`.

'target_compile_definitions' is a CMake command that defines and adds one or more compiler variables to the desired library or executable application. In our case, the variable `MYMATH_SHLIB_EXPORTS` is added for each OS. Depending on the OS, one of the variables `MU_WINDOWS`, `MU_MACOS`, or `MU_LINUX` is added. Now, it's simple to detect the OS type in the code:

```
#ifdef MU_MACOS
    //code for macos
#endif
```

Verifying Exported Symbols

Now it is easy to verify that the `.lib` file has been generated and that the library exports exactly the symbols we want it to export. Using the command `dumpbin /EXPORTS DLL_NAME` (in VS PowerShell, as the command is part of VS), the following output is obtained:

```
Developer PowerShell Visual Studio Community 2022 17.3.32922.545
PS R:\etf02\Debug> dumpbin /EXPORTS myMath.dll
Microsoft (R) COFF/PE Dumper Version 14.33.31630.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file myMath.dll

File Type: DLL

  Section contains the following exports for myMath.dll

    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
           1 ordinal base
           6 number of functions
           6 number of names

    ordinal hint RVA      name

          1    0 000011E5 ??0MyMath@@QEAA@H@Z = @ILT+480(??0MyMath@@QEAA@H@Z)
          2    1 00001195 ??4MyMath@@QEAAAEAV0@$$QEAV0@@Z = @ILT+400(??4MyMath@@QEAAAEAV0@$$QEAV0@@Z)
          3    2 00001037 ??4MyMath@@QEAAAEAV0@AEBV0@@Z = @ILT+50(??4MyMath@@QEAAAEAV0@AEBV0@@Z)
          4    3 0000119F ?add@MyMath@@QEAANN@Z = @ILT+410(?add@MyMath@@QEAANN@Z)
          5    4 000010EB ?mult@MyMath@@QEAANN@Z = @ILT+230(?mult@MyMath@@QEAANN@Z)
          6    5 00001177 ?saberiDvaBroja@@YAHHH@Z = @ILT+370(?saberiDvaBroja@@YAHHH@Z)

  Summary

        1000 .00cfg
        1000 .data
        1000 .idata
        1000 .pdata
        3000 .rdata
        1000 .reloc
        1000 .rsrc
        8000 .text
PS R:\etf02\Debug>
```

Slika 17: Output of `dumpbin /EXPORTS` showing specific exported symbols.

1.9 Using External Libraries

In our previous work, we exclusively developed and used our own libraries, which we compiled and linked within the same project. This simply required adding them with `‘add_dependencies‘` in the `‘CMakeLists.txt‘` of the project where they were to be used, ensuring they were compiled and linked alongside the main project.

However, it is very common in practice to use ready-made libraries (which is one of the fundamental reasons for libraries' existence) that have been pre-compiled and linked (dynamic) on another computer, and whose source code is not available. In such cases, including libraries is done somewhat differently.

The ‘natID‘ framework provides several libraries that form the basis for the lectures and exercises of this course:

- `mainUtils` - This is a mandatory library of the framework. It contains implementations for Unicode string conversion, regional settings for dates and decimal points, local directories, logging, and more.
- `dataProvider` - Contains implementations for working with databases (SQLite, ODBC, SQL Server, Oracle, etc.).
- `netOp` - Contains implementations for intra/internet communication.
- `natGUI` - Contains the framework implementation for working with Graphical User Interfaces (GUI).

Below is a template project that includes the `mainUtils` library. Any subsequent project can be generated by simply copying this project to a new location and changing the project name.

Linking External Libraries with CMake The use of external libraries in CMake is enabled using the `target_link_libraries` command.

- The command format is:

```
target_link_libraries(MODULE_NAME library1 library2 ... libraryN)
```

- This command allows for a variable number of libraries to be specified, thereby including all necessary libraries.
- For example:

```
target_link_libraries(${MY_MODULE} lib/extLib1.lib lib/extLib2)
```

- If you possess debug-enabled versions of libraries, you can use the keywords `debug` or `optimized` before the library name.
- For example:

```
target_link_libraries(${MY_MODULE} debug lib/extLib1D.lib optimized lib/extLib1.lib
lib/extLib2)
```

- Since library extensions depend on the operating system, library names can be assigned to a variable based on the OS, and then included in the linking process via that variable.

An example of setting a variable with the library name, dependent on the OS, for the case of ‘mainUtils‘ from ‘natID‘, is shown below:

```
set(MY_INC ${WORK_ROOT}/Common/Include)

if (WIN32)
    set(PLATFORM_INC "${MY_INC}/Platforms/win")
    if( CMAKE_SIZEOF_VOID_P EQUAL 4 )
        set(MU_LIB_DEBUG "${MY_LIB}/mainUtils32D.lib")
        set(MU_LIB_RELEASE "${MY_LIB}/mainUtils32.lib")
    else()
        set(MU_LIB_DEBUG "${MY_LIB}/mainUtilsD.lib")
        set(MU_LIB_RELEASE "${MY_LIB}/mainUtils.lib")
    endif()
elseif(APPLE)
    set(PLATFORM_INC "${MY_INC}/Platforms/mac")
    set(MU_LIB_DEBUG "${MY_LIB}/mainUtilsD.dylib")
    set(MU_LIB_RELEASE "${MY_LIB}/mainUtils.dylib")
else()
    set(PLATFORM_INC "${MY_INC}/Platforms/linux")
    set(MU_LIB_DEBUG "${MY_LIB}/mainUtilsD.so")
    set(MU_LIB_RELEASE "${MY_LIB}/mainUtils.so")
```

Slika 18: CMake configuration for OS-dependent `mainUtils` library variables.

After this, the variables `MU_LIB_DEBUG` and `MU_LIB_RELEASE` can be used individually or both, depending on the compilation mode.

- For example, if we want to use `MU_LIB_DEBUG` independently of the compilation mode, we do not use the ‘debug‘/‘optimized‘ keywords:

```
target_link_libraries(${MY_MODULE} ${MU_LIB_DEBUG})
```

- If we want to add the `MU_LIB_DEBUG` version in debug mode and `MU_LIB_RELEASE` in release mode, the command would be (with ‘myMath‘ library added first):

```
target_link_libraries(${MY_MODULE} myMath
                      debug ${MU_LIB_DEBUG}
                      optimized ${MU_LIB_RELEASE})
```

For easier work, students are provided with a `Common.cmake` file within the framework, which needs to be included in subsequent projects. `Common.cmake` contains definitions of the most frequently used variables and functions for this course.

1.10 Organizing the Source Code

In practical development, projects often involve hundreds of source code files. Previously, adding source code files to CMake files was a manual, one-by-one process, which can be demanding for large projects. Instead, it's possible to select source code files from a desired location (`src` or another) using pattern matching. CMake provides the `file` command to form variables containing a list of files from a specific location.

- The command `file(GLOB VARIABLE_NAME pathToLocation/*.cpp)` creates a variable `VARIABLE_NAME` that contains all `.cpp` files in the `pathToLocation` directory.
- The command `file(GLOB VARIABLE_NAME pathToLocation/*.h)` creates a variable `VARIABLE_NAME` that contains all `.h` files in the `pathToLocation` directory.

Example:

```
3  file(GLOB MYMATH_SHLIB_TEST_SOURCES  ${CMAKE_CURRENT_LIST_DIR}/src/*.cpp)
4  file(GLOB MYMATH_SHLIB_TEST_INCS  ${CMAKE_CURRENT_LIST_DIR}/src/*.h)
5  file(GLOB MYMATH_SHLIB_TEST_COMMON_INCS  ${CMAKE_CURRENT_LIST_DIR}/../common/include/*.h)
```

Slika 19: Example CMake commands for gathering source and header files.

- Line 3 forms the variable `MYMATH_SHLIB_TEST_SOURCES`, which contains all `.cpp` files within the `src` directory of the executable module `myMathTest`.
- Line 4 forms the variable `MYMATH_SHLIB_TEST_INCS`, which contains all `.h` files within the `src` directory of the executable module `myMathTest`.
- Line 5 forms the variable `MYMATH_SHLIB_TEST_COMMON_INCS`, which contains all `.h` files within the common directory of the entire project.

Collecting and Grouping Source Code

The newly formed CMake variables `MYMATH_SHLIB_TEST_SOURCES`, `MYMATH_SHLIB_TEST_INCS`, and `MYMATH_SHLIB_TEST_COMMON_INCS` are now added to the `myMathTest` project (executable application module) as follows:

```
add_executable(${MYMATH_SHLIB_TEST_NAME} ${MYMATH_SHLIB_TEST_SOURCES} ${MYMATH_SHLIB_TEST_INCS} ${MYMATH_SHLIB_TEST_COMMON_INCS})
```

Slika 20: Adding gathered files to a CMake module.

This drastically accelerates the process of adding code to modules. It eliminates the need for manual, one-by-one additions, as CMake automatically includes all source code files that match the specified pattern.

Note: Header files (`.h`) do not strictly need to be added to the CMake configuration. They are implicitly found by the compiler when included in `.cpp` files. However, in practice, it is highly desirable to have header files visible in the IDE environment. This facilitates easier access to the files and speeds up work during source code editing. Grouping code into folders within the IDE is only possible if all files are explicitly added to the module.

Grouping Source Code within the IDE

After adding all groups of source code to the module, it is possible to organize them within the IDE using the `source_group` command. The `source_group` command works based on directory principles and can create a tree structure just like in an operating system.

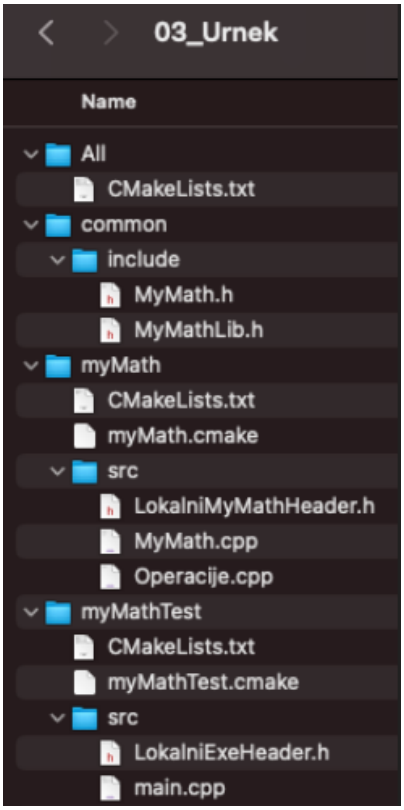
Example for our case:

```
#formatirati foldere u IDE.u
source_group("inc"                  FILES ${MYMATH_SHLIB_TEST_INCS})
source_group("inc\\common"          FILES ${MYMATH_SHLIB_TEST_COMMON_INCS})

source_group("inc\\td"              FILES ${MYMATH_SHLIB_TEST_TD})
source_group("inc\\cnt"             FILES ${MYMATH_SHLIB_TEST_CNT})
source_group("inc\\mu"              FILES ${MYMATH_SHLIB_TEST_MU})
source_group("inc\\fo"              FILES ${MYMATH_SHLIB_TEST_FO})
source_group("inc\\dp"              FILES ${MYMATH_SHLIB_TEST_DP})
source_group("inc\\xml"             FILES ${MYMATH_SHLIB_TEST_XML})

source_group("src"                  FILES ${MYMATH_SHLIB_TEST_SOURCES})
```

Slika 21: Example of using `source_group` to organize files in an IDE.

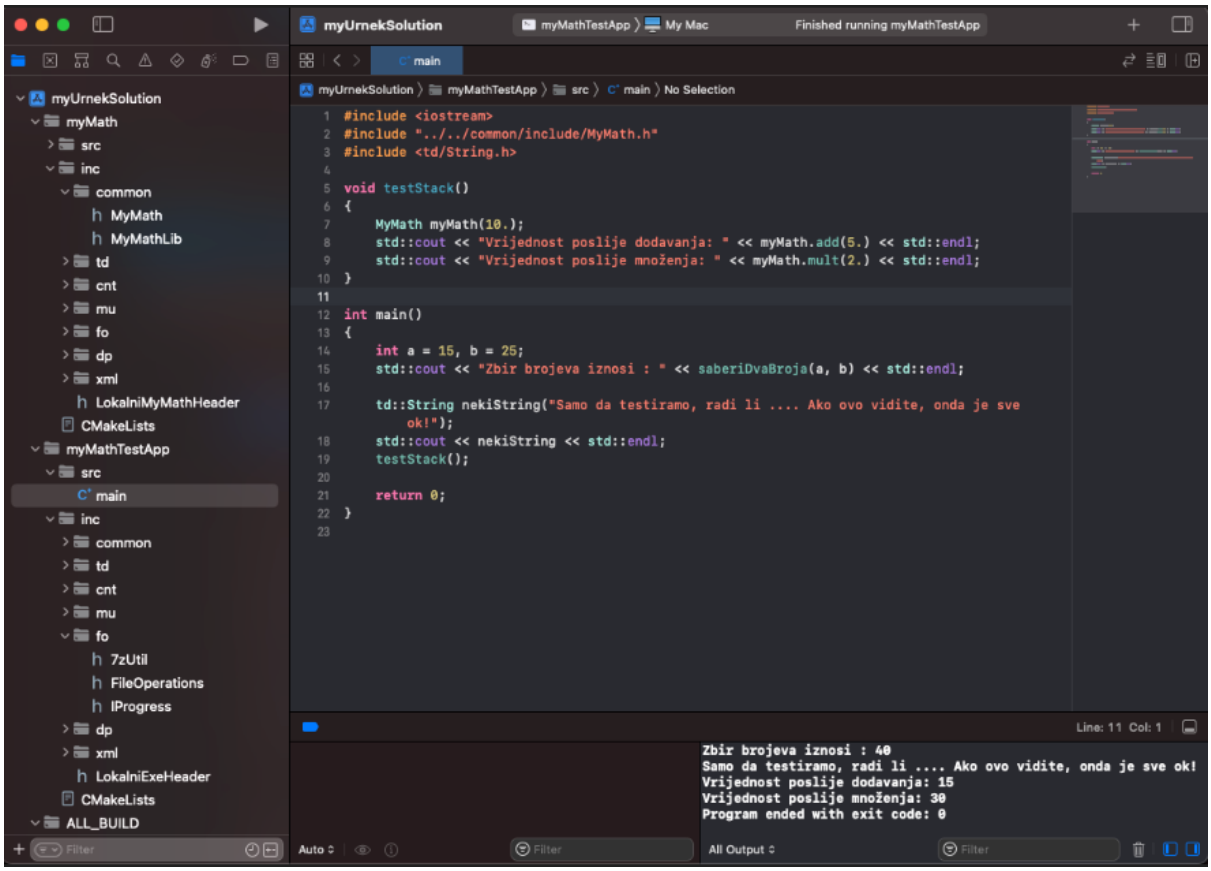
- This creates ‘inc‘ and ‘src‘ directories within the selected IDE.
- Within the ‘inc‘ directory, subdirectories like ‘common‘, ‘td‘, ‘cnt‘, ‘mu‘, ‘fo‘, and ‘dp‘ are created.

This organization significantly simplifies the search for header files. A demonstration of using external libraries and grouping code is provided within the ‘03_Urnek‘ project, specifically in the ‘All‘ subfolder, which needs to be converted using CMake.



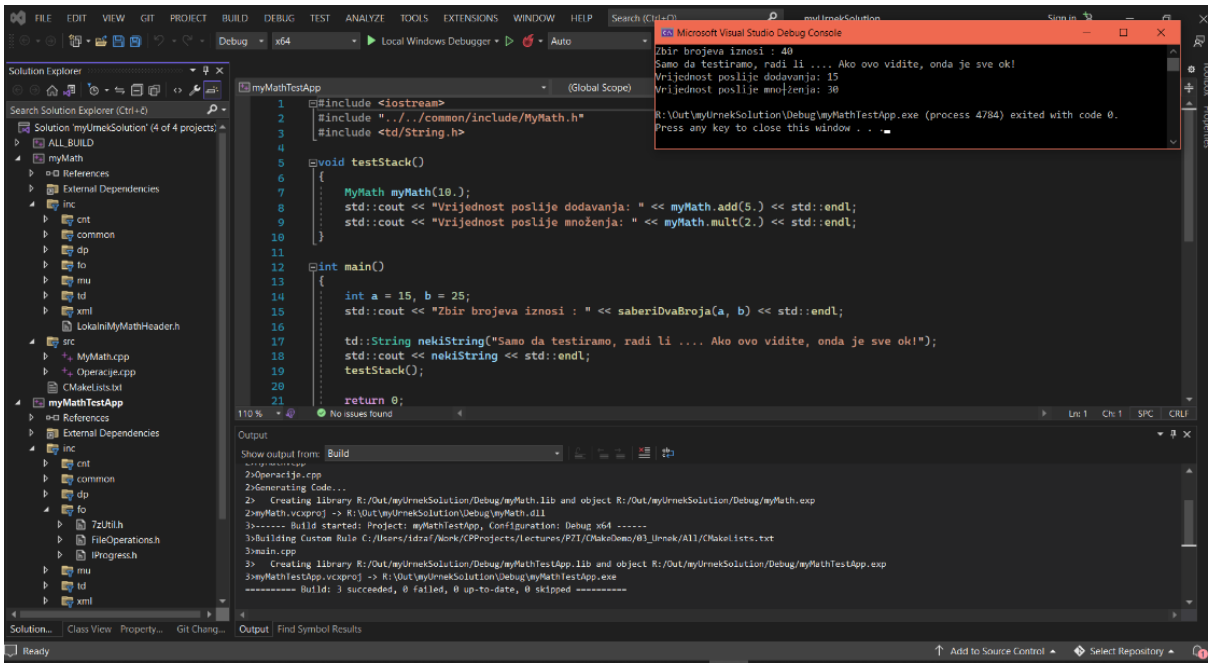
Slika 22: Project strucutre of 03_Urnek.

Project View in Xcode (IDE) with Source Code Folders (Grouping):



Slika 23: Xcode IDE project view with organized source code folders.

Project View in Visual Studio 2022 (IDE) with Source Code Folders (Grouping):



Slika 24: Visual Studio 2022 IDE project view with organized source code folders.

2 Input-Output Operations

2.1 Text Files

Input-output operations that use human-readable text as a method for storing and loading data represent one of the oldest and most widely used approaches.

For input-output operations, various methods can be employed:

- **C-style functions:** `fprintf`, `fscanf`, etc.
- **C++ streams:** `std::ifstream`, `std::ofstream`, and the stream operators « and ».

Advantages:

- Allows for data editing using readily available text editors.
- Simple implementation (for simple models) using stream operators « and ».
- No system architecture problems (if ASCII or UTF-8 encoding is used).
- Data can be used on any system regardless of the OS or processor architecture.

Disadvantages:

- Storing more complex data models becomes very complicated.
- Lack of data description makes editing more complex.
- Slower data writing because each numerical variable requires conversion to text.
- Slower data loading because each numerical variable requires conversion from text to binary format.
- Consumes more disk space.
- Numerical data occupies a variable amount of memory on disk, depending on the length of the converted text.
- Data is written and loaded sequentially.

Demonstration

For this demonstration, we will use the CMake project "`samoExe`", which needs to be copied to a desired location. Create a project for your preferred IDE.

The content of the `main` method (adjust `fileName` as desired):

```
int main()
{
    const char* fileName = "/Volumes/Ramdisk/Test.txt";
    writeData(fileName);
    readData(fileName);
    return 0;
}
```

Slika 25: Example `main` method for text file I/O.

The `writeData` method writes numerical and text data using the streaming operator « to a text file, e.g.:

```
void writeData(const char* fileName)
{
    std::ofstream f;
    f.open(fileName);
    if (f.is_open())
    {
        std::string str("Howdy!");
        float flt = 15.1234f;
        int x = 1234567;
        double pi = std::asin(1.) * 2;
        f << x << " " << flt << " " << str << " " << pi << std::endl;
    }
}
```

Slika 26: Example `writeData` method using stream operator «.

Q: What is the content of the `Test.txt` file?

Content of `Test.txt` The content of the `Test.txt` file is human-readable:

```
1234567 15.1234 Howdy! 3.14159
```

Slika 27: Example content of `Test.txt`.

Loading data using the stream operator » is implemented in the `readData` method:

```
void readData(const char* fileName)
{
    std::ifstream f;
    f.open(fileName);
    if (f.is_open())
    {
        std::string str;
        float flt;
        int x;
        double pi;
        f >> x >> flt >> str >> pi;
        std::cout << x << " " << flt << " " << str << " " << pi << std::endl;
    }
}
```

Slika 28: Example `readData` method using stream operator ».

The expected output on the console (`cout`) is identical to the content in the `Test.txt` file.

```
1234567 15.1234 Howdy! 3.14159
```

Slika 29: Expected console output from reading `Test.txt`.

However, there are at least two problems with this approach.

Q: What are these problems?

Further Discussion on Text File I/O Problems

This method of storing and loading data is very elegant and suitable for tabular numerical data. However, when text data is present, things become complicated.

Q: What would be printed to the console (`cout`) if, instead of the text "Howdy!", the text "Howdy! How are ya?" were written?

- The » operator, when used with a string variable, does not know the length of the string; it reads only up to the first whitespace.
- Consequently, the double number `pi` would then be read from the position where "How are ya?" starts, and it would be assigned a value of zero.

Slika 30: Console output illustrating the problem with reading strings containing spaces.

This problem also exists in binary mode but is easily solved. **Q:** How?

The answer is in the section explaining binary I/O mode.

The second problem that arises is also related to text variables:

Q: What if the text contains a character that does not belong to the ASCII character set?

For this, it is necessary to use one of the Unicode encoding schemes, which further complicates working with text files.

2.2 Binary Files

Binary files operate on the principle of copying the memory location where a variable is stored directly into the file. For input-output operations, the following methods can be used: `fread`, `fwrite`, `fseek`, `read`, `write`, `seek`, etc.

Advantages

- Ability to work with direct access to parts of the data within the file.
- Foundation for implementing relational databases.
- Very fast read and write operations because there is no conversion intermediary step.
- They occupy less space compared to text files.
- Numerical data takes up only as much space as needed to be stored in RAM.
- Simple storage of more complex models.

Disadvantages

- Text editors cannot be used to edit binary files.
- The application that creates the file must also have a read/edit module.
- Can have issues with system architecture:
  - Big/Little Endian:
  - The problem arises if data is saved on a computer with one architecture (e.g., using big-endian) and opened on a computer with another architecture (e.g., little-endian).
  - In this case, all numerical values that have a length greater than 1 byte will be incorrect.

Byte Ordering

Numerical variables that store values greater than 255 require more than one byte, for example:

- `short` - usually 2 bytes.
- `int` - usually 4 bytes.
- `long long` - usually 8 bytes.

Depending on the position of the least (or most) significant byte in a variable, we distinguish architectures with so-called big-endian and little-endian methods. Certain systems support both modes of operation (it is necessary to select one of the two). Data exchange between multiple systems can be resolved by:

- Always recording data in one format (e.g., Java writes all numerical values in big-endian format).
- Or, the file includes the architecture type, and then numerical data is converted if the file is opened (used) on a different architecture.

Big-endian

- SPARC, z/Architecture
- The least significant byte is located at the highest address location.

Little-endian

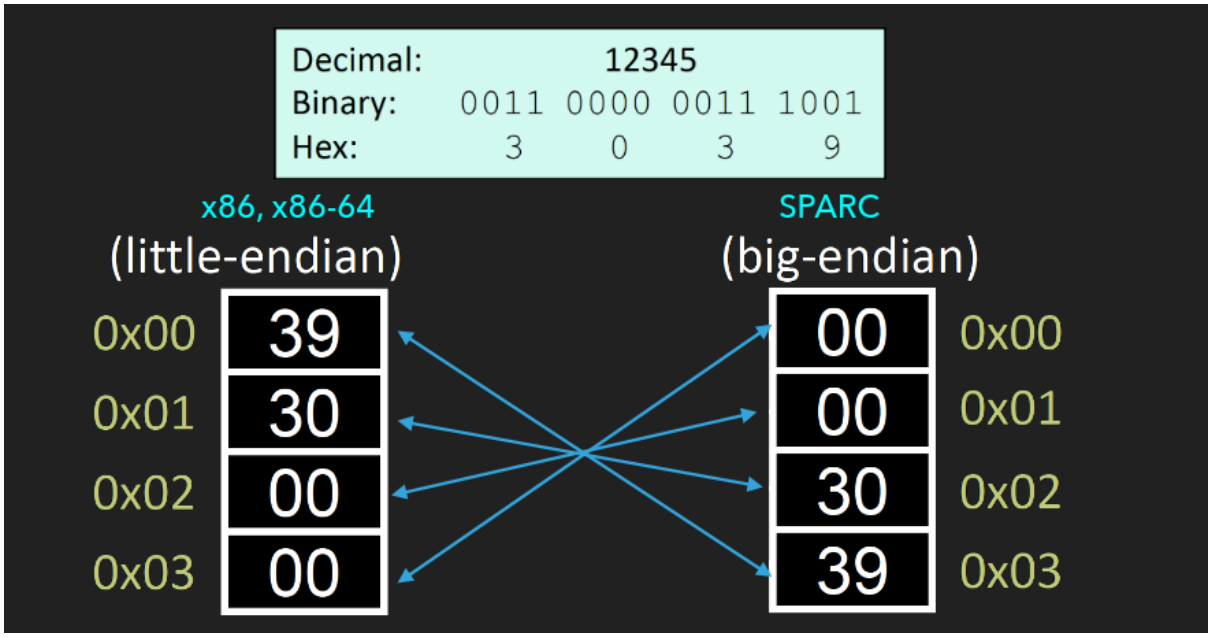
- x86, x86-64
- The least significant byte is located at the lowest address location.

Bi-endian (support both modes of operation)

- ARM, PowerPC
- The operating mode is set by the operating system.

Example

To illustrate, let's observe how the number 12345 (an `int` - 4 bytes) is stored at memory locations from 0x00 to 0x03 on both systems. To more easily observe the bytes within the `int` variable, we will convert the number to hexadecimal (0x 00 00 30 39). Thus, assuming knowledge of the system by which the integer value was saved and detecting the architecture on which the number is being loaded, the problem is solved by a simple byte swap.



Slika 31: Byte ordering example

Demonstration of Binary Mode

To write data in binary mode, it is necessary to open the file for binary mode operations. The `main` function is identical to the example for text operations. Writing data in binary mode for our example is located in the `writeData` method (`write` - method):

```
void writeData(const char* fileName)
{
    std::ofstream f;
    f.open(fileName, std::ios::binary | std::ios::out);
    if (f.is_open())
    {
        std::string str("Howdy! How are you?");
        float flt = 15.1234f;
        int x = 1234567;
        double pi = std::asin(1.) * 2;
        f.write((const char*) &x, sizeof(int));
        f.write((const char*) &flt, sizeof(float));
        size_t strLen = str.length();
        //upisivanje duzine stringa
        f.write((const char*) &strLen, sizeof(size_t));
        //upis sadrzaja stringa
        f.write(str.c_str(), strLen);

        f.write((const char*) &pi, sizeof(double));
        f.flush();
    }
}
```

Slika 32: Writing data using the writeData method

It is important to note that before writing the string content, its length is written. Also, the generated binary file is not human-readable:

Slika 33: Writing data using the writeData method

Reading data in binary mode for our example is located in the `readData` method (`read` - method):

```
void readData(const char* fileName)
{
    std::ifstream f;
    f.open(fileName);
    if (f.is_open())
    {
        std::string str;
        float flt;
        int x;
        double pi;
        f.read((char*) &x, sizeof(int));
        f.read((char*) &flt, sizeof(float));
        //ucitanje duzine stringa
        size_t strLen;
        f.read((char*) &strLen, sizeof(size_t));
        char* tmp = new char[strLen + 1];

        f.read(tmp, strLen);
        tmp[strLen] = '\0';
        str = tmp;
        delete [] tmp;

        f.read((char*) &pi, sizeof(double));

        std::cout << x << " " << flt << " " << str << " " << pi << std::endl;
    }
}
```

Slika 34: Writing data using the writeData method

It can be observed that binary mode requires writing "more code" and is not as elegant as text mode with the $\ll$ and $\gg$ operators.

If this code is executed, the correct output will be displayed in the console:

```
1234567 15.1234 Howdy! How are you? 3.14159
```

Slika 35: Writing data using the writeData method

2.3 NatID Framework

As we have seen so far, C++ does not solve all problems regarding portability. However, C++ is very flexible and allows for extensions.

natID (Native Interface Design) is a framework that provides extensions to C++ libraries in various areas and is divided into multiple namespaces:

- `td` (type definition): Defines new types.

  - `td::String` (with UTF-8, UTF-16, and UTF-32 support), `td::Variant`, etc.

- `mu` (main utils): Application, regionals, logger, etc.
- `fo` (file operations): Creating files with Unicode names, directory search, directory creation, etc.
- `no` (network operations): Internet communications.
- `cnt` (containers): Auxiliary containers and buffers.
- `arch` (archive): Working with binary archives (input-output operations on disks and RAM).
- `dp` (data provider): Working with databases.
- `gui`: Defines a set of classes for user interface development.
- `xml`: SAX and DOM XML parser, XML writer and many more...

2.4 Binary Mode with Class Archive

In the following analysis of input-output operations with binary files, let's consider a more specific problem. Suppose it is necessary to create a dynamic library that generates, saves, and loads data about graphical elements. For simplicity, we will limit the graphical elements to 3 types: rectangle, rounded rectangle, and circle. The dynamic library exports each of these classes.

The CMake project is generated by copying the 'Urnek' CMake project to a new location and changing the module names:

- The `myMath` module becomes `shapeIO`.
- `MYMATH_SHLIB_` becomes `SHAPE_IO_` (find and replace within the CMake file).
- `target_compile_definitions`($MYMATH\_SHLIB\_NAMEPUBLICMYMATH\_SHLIB\_EXPORTS$) changes to `target_compile_definitions`($SHAPE\_IO\_NAMEPUBLICSHAPE\_IO\_EXPORTS$).
- The `myMathTest` module becomes `shapeEditor`.
- `MYMATH_SHLIB_TEST_` becomes `SHAPE_EDITOR_` (find and replace within the CMake file).

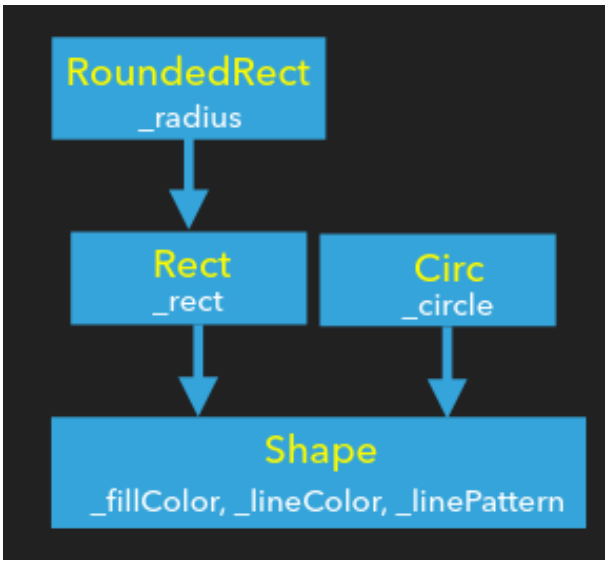
At the `common/include` location, we rename `myMathLib.h` to `shapeIOLib.h` and change its content to:

```
#pragma once

#ifdef MU_WINDOWS
    #ifdef SHAPE_IO_EXPORTS
        #define SHAPE_IO_API __declspec(dllexport)
    #else
        #define SHAPE_IO_API __declspec(dllimport)
    #endif
#else
    #ifdef SHAPE_IO_EXPORTS
        #define SHAPE_IO_API __attribute__((visibility("default")))
    #else
        #define SHAPE_IO_API
    #endif
#endif
```

Slika 36: Content of `shapeIOLib.h`

UML (Unified Modeling Language) class diagrams can be done in various ways. One proposal is shown in the image below. The executable application `shapeEditor` has a `Model` class which holds a list of pointers to `Shapes`.

Important: The principle of OO design is based on extracting common attributes and methods into a base class. The process continues according to the same principle in "inheriting" classes.



Slika 37: UML Class Diagram for Shape Hierarchy

The base class `Shape` defines the interface and stores common attributes for fill color, line color, and line type:

```
#pragma once
#include "ShapeIOLib.h"
#include <td/Types.h>
#include <arch/ArchiveIn.h>
#include <arch/ArchiveOut.h>

class SHAPE_IO_API Shape
{
public:
    enum class Type : td::BYTE {Rect=0, RoundedRect, Circ};
protected:
    td::ColorID _fillColor;
    td::ColorID _lineColor;
    td::LinePattern _linePattern;
public:
    Shape(td::ColorID fillColor, td::ColorID lineColor, td::LinePattern lp);
    Shape();

    virtual ~Shape();
    virtual Shape::Type getType() const = 0;
    virtual void show(std::ostream& o) const;
    virtual void save(arch::ArchiveOut& aOut) const;
    virtual void load(arch::ArchiveIn& aIn);
};
```

Slika 38: Shape Class Definition

- The `show` method displays attributes on the graphical console.
- The `save` method uses the `ArchiveOut` class for simple writing of `Shape`'s content to a binary file.
- The `load` method uses the `ArchiveIn` class for simple loading of `Shape`'s content from a binary file.

The `Rect` class inherits common attributes from the `Shape` class and adds information about the rectangle (using the template class `td::RectNormalized`):

```cpp
#pragma once
#include "Shape.h"
#include <td/RectNormalized.h>

typedef td::RectNormalized<double> Rectangle;

class SHAPE_IO_API Rect : public Shape
{
protected:
    Rectangle _rect;
public:
    Rect(const Rectangle& r, td::ColorID fillColor, td::ColorID lineColor,
        td::LinePattern lp);
    Rect();
    virtual Shape::Type getType() const;
    virtual void show(std::ostream& o) const;
    virtual void save(arch::ArchiveOut& aOut) const;
    virtual void load(arch::ArchiveIn& aIn);
```

Slika 39: Rect Class Definition

The `RoundedRect` class inherits common attributes from the `Rect` class and adds information about the corner radius:

```cpp
#pragma once
#include "Rect.h"


class SHAPE_IO_API RoundedRect : public Rect
{
protected:
    double _radius;
public:
    RoundedRect(const Rectangle& r, double radius, td::ColorID fillColor,
        td::ColorID lineColor, td::LinePattern lp);
    RoundedRect();

    virtual Shape::Type getType() const;
    virtual void show(std::ostream& o) const;
    virtual void save(arch::ArchiveOut& aOut) const;
    virtual void load(arch::ArchiveIn& aIn);
};
```

Slika 40: RoundedRect Class Definition

Since the `Circ` class does not share common properties with the `Rect` and `RoundedRect` classes, it inherits common attributes from the `Shape` class and adds information about the circle's center and radius (utilizing the template class `td::Circle`):

```
#pragma once
#include "Shape.h"
#include <td/Circle.h>

typedef td::Circle<double> Circle;

class SHAPE_IO_API Circ : public Shape
{
protected:
    Circle _circle;
public:
    Circ(const Circle& c, td::ColorID fillColor, td::ColorID lineColor,
        td::LinePattern lp);
    Circ();
    virtual Shape::Type getType() const;
    virtual void show(std::ostream& o) const;
    virtual void save(arch::ArchiveOut& aOut) const;
    virtual void load(arch::ArchiveIn& aIn);
```

Slika 41: Circ Class Definition

In addition, it is necessary to implement a container class that will utilize the principle of polymorphism and place all graphical elements (`Shape`s) into a single vector.

- This allows for simple management of the drawing order of graphical elements (in some of the upcoming lectures).
- The name of this class is `Model`.

The task of the `Model` class is actually to:

- Manage the lifecycle of graphical elements.
- It can create a graphical element and can delete it.
- To call specific operations on `Shape` class objects (`save`, `load`, `show`, `draw`, etc.).

The basic idea of this project is to demonstrate the fundamental principles of OO design with a simple example:

- Class diagram and how to create it based on the principles of common attributes and functions.
- Polymorphism (virtual functions).

Data input is performed from the console, and saving and loading are done using binary files. In some of the upcoming lectures, input will be done using a graphical interface, and the ability to change attributes and draw graphical elements will be added. (Stay tuned. :-))

The main `main` function of our "editor" displays a menu with certain options:

```
1  #include <iostream>
2  #include <td/String.h>
3  #include <iostream>
4  #include "Model.h"
5  int main()
6  {
7      Model model;
8      size_t odabir = 10;
9      while (odabir > 0)
10     {
11         std::cout << "1. Kreiraj Rect" << td::endl;
12         std::cout << "2. Kreiraj RoundedRect" << td::endl;
13         std::cout << "3. Kreiraj Circ" << td::endl;
14         std::cout << "4. Snimi" << td::endl;
15         std::cout << "5. Ucitaj" << td::endl;
16         std::cout << "6. Prikazi model na konzoli" << td::endl;
17         std::cout << "0. Kraj" << td::endl;
18         std::cout << "Unesize vas izbor:";
19         std::cin >> odabir;
20         if (odabir > 6){std::cout << "Pogresna meni opcija." << td::endl; continue; }
21         switch (odabir)
22         {
23             case 1: model.addRect();  break;
24             case 2: model.addRoundedRect(); break;
25             case 3: model.addCircle(); break;
26             case 4: model.save("/Volumes/RAMDisk/Shapes.bin"); break;
27             case 5: model.load("/Volumes/RAMDisk/Shapes.bin"); break;
28             case 6: model.show(std::cout); break;
29         }
30     }
31     return 0;
32 }
```

Slika 42: Main Function Menu

- An object of the `Model` class is created on line 7.
- From lines 11 to 18, the menu is displayed.
- The first three options are used to create graphical elements (lines 23 - 25), currently via the console.
- Options 4 and 5 are used for saving and loading the model using binary (file) mode (lines 26 and 27).
- Printing the model is executed with the sixth option (line 28), currently via the console.

Saving The `arch` namespace is used for all classes in the group that perform data archiving (to disk or memory). The `ArchiveOut` class implements the '«' operators in such a way that saving can be performed to a binary file or to a memory buffer. In our case, it saves to a file, so to construct an `ArchiveOut` object, it is necessary to pass a `FileSerializerOut` object, which is tasked with writing data to the file.

- In the case of saving data to memory, it is necessary to use an object of the appropriate class (the second parameter of the `ArchiveOut` constructor).

The `Model` class implements the `save` method by instantiating an `ArchiveOut` object and then iteratively passing through all elements within the `_shapes` container, calling the virtual `save` method. Since it is necessary to know how many graphical elements are saved in the file for the loading process, it is necessary

to write not only the attributes of the graphical elements but also the total number of elements and the type of each element.

- The element type is obtained via the `Shape::getType` method.

Using the `ArchiveOut` class, a simple implementation of data saving is achieved (`Method::save`).

```
102     bool save(const char* fileName) const
103     {
104         arch::FileSerializerOut fs;
105         if (!fs.open(fileName))
106             return false;
107         arch::ArchiveOut ar("ETFA", fs);
108         td::UINT4 nShapes = _shapes.size();
109         try
110         {
111             ar << nShapes;
112             for (auto pShape : _shapes)
113             {
114                 td::BYTE shType = (td::BYTE) pShape->getType();
115                 ar << shType;
116                 pShape->save(ar);
117             }
118         }
119         catch(...)
120         {
121             return false;
122         }
123         return true;
124     }
```

Slika 43: Model Save Implementation

- The construction of the `ar` object (archive for saving data) is realized on lines 104-107.
- It is necessary to pay attention that the archive constructor has two parameters:
  - The first parameter of 4 characters is a "magic word" and serves to verify during writing whether it is actually an archive of the desired type.
  - This prevents parsing of files that do not contain our model.
  - It is arbitrarily chosen for the model type.
  - When loading, an identical magic word must be specified.
- After this, the total number of elements is written once, and then for each element, its type and attributes are written (`pShape->save`).
- `RoundedRect::save` saves local attributes and calls the base class `Rect` to do the same.

Slika 44: Rounded rectangle save method

- `Rect::save` saves local attributes and calls the base class `Shape` to do the same.

Slika 45: Rectangle save method

- `Shape::save` saves local attributes.

Slika 46: Shape save method

- `Circ::save` saves local attributes and calls the base class `Shape` to do the same.

Slika 47: Circle save method

We can observe that the saving of attributes itself is very simple, which speeds up implementation and makes the code more readable in the case of a large number of classes.

Loading The `ArchiveIn` class implements the '»' operators in such a way that data loading can be performed from a binary file or from a memory buffer. In our case, data is loaded from a file, so to construct an `ArchiveIn` object, it is necessary to pass a `FileSerializerIn` object, which is tasked with loading data.

- In the case of loading from memory, it is necessary to use an object of the appropriate class (the second parameter of the `ArchiveIn` constructor).

The `Model` class implements the `load` method by instantiating an `ArchiveIn` object.

- `ArchiveIn` checks whether the magic word in the file matches the required one. If not, an exception is generated, and the loading process terminates.
- After that, the total number of elements stored in the file (`nShapes`) is loaded.
- A loop that runs `nShapes` times is opened.
- After loading the element type, an object of the corresponding type is generated using the default constructor, and then the virtual `load` method is called.

Using the `ArchiveIn` class, a simple implementation of data loading is achieved (`Method::load`).

```
126     bool load(const char* fileName){
127         clean();
128         arch::FileSerializerIn fs;
129         if (!fs.open(fileName))
130             return false;
131         arch::ArchiveIn ar(fs);
132         ar.setSupportedMajorVersion("ETFA");
133         td::UINT4 nShapes = 0;
134         try{
135             ar >> nShapes;
136             for (size_t i=0; i<nShapes; ++i) {
137                 td::BYTE shType;
138                 ar >> shType;
139                 Shape::Type st = (Shape::Type) shType;
140                 Shape* pShape = nullptr;
141                 switch (st) {
142                     case Shape::Type::Rect: pShape = new Rect; break;
143                     case Shape::Type::Circ: pShape = new Circ; break;
144                     case Shape::Type::RoundedRect: pShape = new RoundedRect; break;
145                 }
146                 assert(pShape);
147                 pShape->load(ar);
148                 _shapes.push_back(pShape);
149             }
150         }
151         catch(...){
152             return false;
153         }
```

Slika 48: Model Load Implementation

- Before loading the model from the file, existing elements need to be deleted (linija 126).
- The construction of the `ar` object (archive for saving data) is realized on lines 128-132.
- The magic word is entered on line 132.

- After this, the total number of elements is written once, and then for each element, its type is loaded. Based on this information, the corresponding object is instantiated, and the `load` method is called.
- After this, the element is added to the `_shapes` container.
- `RoundedRect::load` loads its own attributes and calls the base class `Rect` to do the same.

```
30  void RoundedRect::load(arch::ArchiveIn& aIn)
31  {
32      aIn >> _radius;
33      Rect::load(aIn);
34  }
```

Slika 49: Rounded rectangle load method

- `Rect::load` loads its own attributes and calls the base class `Shape` to do the same.

```
30  void Rect::load(arch::ArchiveIn& aIn)
31  {
32      aIn >> _rect;
33      Shape::load(aIn);
34  }
```

Slika 50: Rectangle load method

- `Shape::load` loads local attributes.

```
30  void Shape::load(arch::ArchiveIn& aIn)
31  {
32      arch::EnumLoader<td::LinePattern> lnPattern(_linePattern,
            td::LinePattern::NA, td::LinePattern::Solid);
33      aIn >> _fillColor >> _lineColor >> lnPattern;
34  }
```

Slika 51: Shape load method

- `Circ::load` loads local attributes and calls the base class `Shape` to do the same.

```
30 void Circ::load(arch::ArchiveIn& aIn)
31 {
32     aIn >> _circle;
33     Shape::load(aIn);
34 }
```

Slika 52: Circlee load method

We can observe that the loading of attributes (data) itself is very simple, which speeds up implementation and makes the code more readable in the case of a large number of classes.

Unicode

Most computer science students are familiar with the ASCII character encoding scheme. This was the most widespread encoding for over forty years. ASCII encoding maps characters to 7-bit integers, using a range from 0 to 127 to represent 94 printable characters, 33 control characters, and a space. Since a byte was used to store a character, the eighth bit of the byte was filled with a zero (0).

The problem with ASCII code is that it does not provide a way to encode characters from other scripts, such as Cyrillic or Greek. In 1989, to overcome this problem, the International Organization for Standardization (ISO) began work on a universal, comprehensive character code standard. In 1990, a draft standard (ISO 10646) called the Universal Character Set (UCS) was introduced. UCS was designed as a superset of all other character set standards, providing backward compatibility with other character sets.

At the same time, the Unicode Project, which was a consortium of private industry partners, was working on its own, independent universal character encoding. In 1991, the Unicode Project and ISO decided to collaborate to avoid creating two different character encodings. The result was that the code table developed by the Unicode Consortium (as it is now called) satisfied the original ISO 10646 standard. Over time, the two groups continued to modify their respective standards, but they always remained compatible. Unicode adds new characters over time but always contains the character set defined by ISO 10646-x.

UTF-32, UTF-16, UTF-8 Since code points (characters) are represented by 4 bytes, it would be logical for each character to be represented by 4 bytes. This is the so-called UTF-32 encoding scheme (UTF stands for "UCS (Unicode) Transformation Format"). The first problem with UTF-32 is its dependence on processor architecture (endianness). The second problem with UTF-32 is that it occupies unnecessarily many bytes for most alphabets; most can be saved using 2 bytes.

UTF-16 provides a solution to the second (mentioned above) problem. However, due to big/little endian CPU architectures, there are two versions: UTF-16BE and UTF-16LE. The UTF-8 (or multi-byte) approach solves both problems. The complete interface in `natID` is implemented for UTF-8. Where conversion to UTF-16 (Windows) is needed, the `natID` framework performs it automatically. Therefore, instead of `std::string`, it is necessary to use `td::String`.

UTF-8 The most widespread Unicode encoding as byte sequences is UTF-8, invented by Ken Thompson in 1992. In UTF-8, characters are encoded from one to six bytes. In other words, the number of bytes used varies with the character (UTF-16 is also variable). UTF-8 uses the following scheme for encoding

Unicode code points:

- Characters U+0000 to U+007F (i.e., ASCII characters) are simply encoded as bytes 0x00 to 0x7F.
- All UCS characters greater than U+007F are encoded as a sequence of two or more bytes, each of which has the most significant bits set.
- This means that no ASCII byte can appear as part of any other character, as ASCII characters are the only characters whose leading bit is 0.
- The first byte of a multi-byte sequence representing a non-ASCII character is always in the range 0xC0 to 0xFD and indicates how many bytes follow for this character. Specifically, it is one of 110xxxxxx, 1110xxxx, 11110xxx, 111110xx, and 1111110x, where x's can be 0 or 1. The number of 1-bits that follow the first 1-bit up to the next 0-bit is the number of bytes in the rest of the sequence.
- All subsequent bytes in a multi-byte sequence begin with two bits 10 and are in the range 0x80 to 0xBF.

The foregoing implies that UTF-8 sequences must be of the following forms in binary, where x's represent bits from the code point, with the leftmost x-bit being its most significant bit:

- `0xxxxxxx`
- `110xxxxx 10xxxxxx`
- `1110xxxx 10xxxxxx 10xxxxxx`
- `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
- `111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
- `1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

From the above, it can be seen that bytes 0xFE and 0xFF are never used in UTF-8 encoding.

XML

XML is an acronym for Extensible Markup Language. It is a text-based markup language derived from Standard Generalized Markup Language (SGML). Additional self-describing tags (markup) are introduced into a text document to improve understanding and flexibility of the document. XML tags identify data and are used to store and organize data rather than to specify how to display it (unlike HTML tags which are used to display data).

There are three important characteristics of XML that make it useful in various systems and solutions:

- **XML is extensible:** XML allows you to create your own self-describing tags or languages that suit your application.
- **XML carries data, it does not present it:** XML allows data to be stored regardless of how it will be presented.
- **XML is a public standard:** XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

XML is a self-describing (markup) language that defines a set of rules for encoding documents in a format that is readable by both humans and machines. Markup is information added to a document that enhances its meaning in specific ways by identifying parts and their interrelationships. A markup language is a set of symbols that can be placed in a document's text to delimit and label parts of that document.

To better understand the meaning of this, let's assume we want to implement a system that sends a message consisting of its priority and content. Using a markup language (XML) would format the message as follows:

```
<message>
    <priority>Hitno</priority>
    <content>Moraš učiti inače nećeš položiti predmet.</content>
</message>
```

Slika 53: XML Message Example

This snippet includes markup symbols or tags such as `<message></message>`, `<priority></priority>`, `<content>` and `</content>` which do not belong to the message itself but facilitate easier parsing (understanding) of the document.

- The `<message>` and `</message>` tags mark the beginning and end of the XML code fragment (message).
- The `<priority>` and `</priority>` tags denote the part of the message related to its priority.
- The `<content>` and `</content>` tags denote the part of the message related to its content (text).

XML Syntax An XML document (file) must begin with a declaration: `<?xml version="1.0" encoding="UTF-8"?>`

Rules for XML declaration:

- The XML declaration is case-sensitive and must start with "`<?xml>`" where "xml" is written in lowercase.
- If the document contains an XML declaration, it must strictly be the first statement of the XML document.
- The HTTP protocol can change the encoding value specified in the XML declaration.

An XML file is structured from several XML-elements, which are called XML-nodes or XML-tags. XML-element names are enclosed in angle brackets `< >` as shown in the message example: `<message>`

XML Syntax for Nodes Every node must have start and end tags: `<message> ... text... </message>` A node can be empty (i.e., without any content): `<message/>` Nodes can be nested. The depth of nesting has no limit. An XML-node can contain multiple XML-nodes as its children, but subordinate elements must not overlap. That is, an element's end tag must have the same name as the last unmatched start tag.

Example of incorrect nesting:

```
<message>
<priority>Hitno</priority>
<content>Moraš učiti inače nećeš položiti predmet.
</message>
</content>
```

Slika 54: Incorrect XML Nesting Example

Because of this, "indentation" of nested elements is recommended.

An XML document can have only one root node. For example, the following XML document is not valid because elements `x` and `y` appear at the top level without a root element:

```
<x>...</x>
<y>...</y>
```

Slika 55: Invalid XML Document (Multiple Roots)

The following example shows a well-formed XML document (a unique root node has been introduced):

```
<root>
    <x>...</x>
    <y>...</y>
</root>
```

Slika 56: Valid XML Document (Single Root)

Nodes are case-sensitive:

- The `<message>` node is different from the `<Message>` node.
- This also applies to attribute names.

Nodes can have attributes in addition to content.

```
<message id="3700">Sadržaj</message>
```

Slika 57: XML Node with an Attribute

`id` is an attribute. A node is correctly opened and closed if it only has attributes in the following way:

```
<message id="3700" txt="Sadržaj"/>
```

Slika 58: Empty XML Node with Attributes

`id` and `txt` are attributes. A node cannot have two attributes with the same name. The text content of a node can be anything encoded with the specified encoding, with the exception of reserved symbols.

XML Reserved Symbols For defining markup, XML uses the symbols `<`, `>`, `&`, `'` `(apostrophe)`, and `"` `(quotation mark)`. These symbols are reserved and cannot be used for other purposes. If the values of node content and attribute values contain a reserved symbol, they are replaced with the following character sequences:

- `<` is replaced with `<`
- `>` is replaced with `>`
- `&` is replaced with `&`
- `'` is replaced with `&apos;`
- `"` is replaced with `&quot;`

If XML file loading and saving operations are performed via the `natID` framework, this conversion is done automatically. A comment node can be entered into an XML document in the following way: `<!- This is a comment ->`

Demonstration of Working with XML through the natID Framework

Let's assume we want to create an XML file that writes a comment and two messages with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<Messages>
<!--Ovo je samo demonstracija.
Poruke se upisuju koristenjem petlje,a podaci se uzimaju iz nekog niza strukture podataka...-->
    <Message id="100" date="2022-10-31" time="06:26:50.5237" priority="High" severity="4.8">Ovo je prva poruka!</Message>
    <Message id="101" date="2022-11-15" time="06:26:50.6305" priority="Low" severity="1.5">Ovo je druga poruka</Message>
</Messages>
```

Slika 59: Example Message Content for XML

When writing, the current system date and time should be taken. The date of the second message should be 15 days after the first. After writing the messages and creating the XML file, it is necessary to parse it and print the attribute values of all messages (`Message` nodes) as well as the content of the messages to the console. The complete implementation is based on the "samoExe" CMake project (copy-paste-edit). The implementation (source code) is provided in the "04_XmlIntro" directory of this lecture supplement.

- Use `td::Date` and `td::Time` classes for date and time manipulations.

XML Writer To create XML documents, it is necessary to use the `xml::Writer` class. Depending on the chosen constructor, it can create XML documents in files or in memory. The `date` and `time` attribute values are taken at the beginning of saving data to the XML file (`date.now()`, `time.now()`):

```
8  void writeXml(const td::String& fileName)
9  {
10     td::Date date;
11     date.now();
12     td::Time time;
13     time.now();
```

Slika 60: XML Writer Date and Time Initialization

After that, an object `w` of the `Writer` class is instantiated:

```
15      xml::Writer w(fileName);
16      w.startDocument();
```

Slika 61: XML Writer Instantiation

The `startDocument` method writes the XML declaration (`<?xml version="1.0" encoding="utf-8"?>`). The `startNode` method begins a new node. The first generated node is also the root node:

```
17      w.startNode("Messages");
```

Slika 62: XML Writer Starting Root Node

The `comment` method is used for writing comments:

```
19      w.comment("Ovo je samo demonstracija.\nPoruke se upisuju koristenjem petlje,"
20                "a podaci se uzimaju iz nekog niza strukture podataka...");
```

Slika 63: XML Writer Comment Method

XML Writer

Writing a message is done as follows:

```
21      //prva poruka
22      w.startNode("Message");
23      int messageID = 100;
24      w.attribute("id", messageID);
25      w.attribute("date", date);
26      w.attribute("time", time);
27      w.attribute("priority", "High");
28      float severity = 4.8;
29      w.attribute("severity", severity);
30      w.nodeString("Ovo je prva poruka!");
31      w.endNode();
```

Slika 64: XML Writer Message Writing Example

- Line 22 opens a new node (i.e., writes `<Message>` to the XML file).
- Writing attributes (all types: `int`, `float`, `real`, `date`, `time`, `string`, etc.) is done using the `attribute` method.
- The content (text) of nodes, or the message content in this case, is written using the `nodeString` method.
- This example shows how both attributes and content of an XML node are written.
- The `endNode` method closes the last open node. In the case of the code shown on line 31, the writer will write `</Message>` to the XML file.
- The second message is written in a similar way, or multiple messages could be written similarly from a program loop.
- Closing all open nodes (including the root node) is done using the `endDocument` method:

```
49      //zatvori sve otvorene cvorove
50      w.endDocument();
51  }
```

Slika 65: XML Writer End Document

XML SAX and DOM Parser For parsing documents, the template classes `xml::SAXParser` and `xml::DOMParser` can be used.

- **SAX parser (Simple API for XML)** generates events for each opening and closing of a new node, attributes, and node content. It works efficiently with memory because it sequentially passes through the XML document and "notifies" about the content.
- **DOM parser (Document Object Model)** is a parser that parses the complete document and provides information about the correctness of the parser. It internally uses a SAX parser.
- The DOM parser can parse documents loaded from a file or from memory (`xml::FileParser` or `xml::MemoryParser`).

In all subsequent activities, only the DOM `FileParser` will be used.

XML FileParser The loading process begins by parsing the XML document (the `parseFile` method). If the XML document does not exist or is incorrectly formatted, the method returns `false`:

```
53  void readXml(const td::String& fileName)
54  {
55      xml::FileParser parser;
56      if (!parser.parseFile(fileName))
57      {
58          std::cout << "Ne mogu parsirati fajl: " << fileName << td::endl;
59          return;
60      }
```

Slika 66: XML FileParser: `parseFile` Method

After this, it is necessary to position on the root node (the `getRootNode` method):

```
62      auto root = parser.getRootNode();
63      if (root->getName().cCompare("Messages") != 0)
64      {
65          std::cout << "Pogresan xml fajl: " << fileName << td::endl;
66          return;
67      }
```

Slika 67: XML FileParser: `getRootNode` Method

Line 63 demonstrates the use of a node name and its verification. In our example, if the root node is different from "Messages," then we have opened the wrong type of XML document.

XML FileParser `xml::FileParser` (and `MemoryParser`) work on the principle of iterators over nodes and exclusively with UTF-8. After detecting the first child of the root node (the `getChildNode` method on line 69), the iterator value is simply incremented (the `++` operator on line 86) as long as it has a valid value (the `isOk` method - line 70):

```
69      auto message = root.getChildNode("Message");
70      while (message.isOk())
71      {
72          int msgID = 0;
73          message.getAttribValue("id", msgID);
74          td::String strPriority;
75          message.getAttribValue("priority", strPriority);
76          float severity;
77          message.getAttribValue("severity", severity);
78          td::Date date;
79          message.getAttribValue("date", date);
80          td::Time time;
81          message.getAttribValue("time", time);
82          td::String strMsgContent = message->getValue();
83
84          std::cout << "Message id=" << msgID << ", priority=" << strPriority << ", severity=" << severity
85              << ", date:" << date << ",time=" << time << ", txt=" << strMsgContent << td::endl;
86          ++message;
87      }
88  }
```

Slika 68: XML FileParser: Iteration through Child Nodes

- Retrieving values of almost all attribute types is implemented via the `getAttribValue` method (operator `.`).
- Retrieving the content (text) of a node is implemented via the `getValue` method (operator `->`).

Flexibility of XML (Demonstration with Classes)

To demonstrate the simplicity and flexibility of XML input-output operations, let's look at what needs to be done for our `shapeEditor` to be able to save and load its model into XML files.

The following methods need to be added to the `Shape` class and other classes (`Rect`, `RoundedRect`, and `Circ`):

```
virtual void save(xml::Writer& w) const;
virtual void load(const xml::FileParser::node_iterator& node);
```

Slika 69: XML Methods Added to Shape Class

Since the `Model` class contains pointers to all `Shapes`, it has implemented the `saveXml` method:

```
bool saveXml(const td::String& fileName) const
```

Slika 70: Model's `saveXml` Method

and the `loadXml` method:

```
bool loadXml(const td::String& fileName)
```

Slika 71: Model's `loadXml` Method

Saving `Model::saveXml` is tasked with:

- Creating an XML document (line 171).
- Writing the XML declaration (line 172).
- Creating the root node (line 173).

After that, it iterates through all graphical elements (through `_shapes`), which is shown on lines 175-195.

- Lines 180-192 create corresponding nodes for `Rect`, `RoundedRect`, and `Circ`.
- The responsibility for writing XML data (attributes) is the task of the corresponding class (line 193).
- The XML document is correctly closed by calling `endDocument` (line 196).

```
169     bool saveXml(const td::String& fileName) const
170     {
171         xml::Writer w(fileName);
172         w.startDocument();
173         w.startNode("Shapes");
174 
175         for (auto pShape : _shapes)
176         {
177             auto st = pShape->getType();
178             switch (st)
179             {
180                 case Shape::Type::Rect:
181                     w.startNode("Rect");
182                     break;
183                 case Shape::Type::RoundedRect:
184                     w.startNode("RoundedRect");
185                     break;
186                 case Shape::Type::Circ:
187                     w.startNode("Circ");
188                     break;
189                 default:
190                     assert(false);
191                     return false;
192             }
193             pShape->save(w);
194             w.endNode();
195         }
196         w.endDocument();
197         return true;
198     }
```

Slika 72: Model's `saveXml` Implementation

- `RoundedRect::save` saves local attributes and calls the base class `Rect` to do the same.

```
36  void RoundedRect::save(xml::Writer& w) const
37  {
38      w.attribute("r", _radius);
39      Rect::save(w);
40  }
```

Slika 73: RoundedRect save method

- `Rect::save` saves local attributes and calls the base class `Shape` to do the same.

```cpp
void Rect::save(xml::Writer& w) const
{
    w.attribute("x", _rect.left);
    w.attribute("y", _rect.top);
    double width = _rect.width();
    w.attribute("width", width);
    double height = _rect.height();
    w.attribute("height", height);
    Shape::save(w);
}
```

Slika 74: Rectangle save method

- `Shape::save` saves local attributes.

```cpp
void Shape::save(xml::Writer& w) const
{
    w.attribute("fillColor", _fillColor);
    w.attribute("lineColor", _lineColor);
    w.attribute("linePattern", _linePattern);
}
```

Slika 75: Shape save method

- `Circ::save` saves local attributes and calls the base class `Shape` to do the same.

```cpp
void Circ::save(xml::Writer& w) const
{
    w.attribute("x", _circle.center.x);
    w.attribute("y", _circle.center.y);
    w.attribute("r", _circle.r);
    Shape::save(w);
}
```

Slika 76: Circle save method

Result of Saving An example of the content of an XML file using the previous code:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Shapes>
    <Rect x="100" y="100" width="60" height="40" fillColor="White" lineColor="Black" linePattern="Solid"/>
    <Circ x="300" y="300" r="80" fillColor="White" lineColor="Black" linePattern="Solid"/>
    <Rect x="200" y="250" width="300" height="400" fillColor="White" lineColor="Black" linePattern="Solid"/>
    <Circ x="350" y="350" r="60" fillColor="White" lineColor="Black" linePattern="Solid"/>
    <RoundedRect r="15" x="400" y="400" width="100" height="70" fillColor="White" lineColor="Black" linePattern="Solid"/>
    <Rect x="1" y="100" width="100" height="150" fillColor="White" lineColor="Black" linePattern="Solid"/>
    <RoundedRect r="1" x="1" y="200" width="200" height="5" fillColor="White" lineColor="Black" linePattern="Solid"/>
</Shapes>
```

Slika 77: Example of Saved XML File Content

- The name of the root node is `Shapes`.
- The root node has multiple different branches (children) or nested nodes:
  - `Rect`,
  - `RoundedRect`, and
  - `Circ`.

- Each of the nested nodes contains all the necessary information for "reincarnation" or creation in the computer's working memory.

Loading `Model::loadXml` is tasked with:

- Parsing the XML document (lines 202-204).
- Taking and checking the root node (lines 206-208).
- Deleting previously created elements if they exist (line 211).
- Selecting the first child node regardless of its name (`getChildNode` without the node name parameter).
- The process of iterating through XML nodes, creating instances of graphic objects, and storing them in the `_shapes` container is given on lines 214-223.

```
200     bool loadXml(const td::String& fileName)
201     {
202         xml::FileParser parser;
203         if (!parser.parseFile(fileName))
204             return false;
205
206         const auto& root = parser.getRootNode();
207         if (root->getName().cCompare("Shapes") != 0)
208             return false;
209
210         //delete previous model entries
211         clean();
212         auto it = root.getChildNode();
213
214         while (it.isOk())
215         {
216             Shape* pShape = nullptr;
217             if (it->getName().cCompare("Rect") == 0)
218                 pShape = new Rect;
219             else if (it->getName().cCompare("RoundedRect") == 0)
220                 pShape = new RoundedRect;
221             else if (it->getName().cCompare("Circ") == 0)
222                 pShape = new Circ;
223             if (!pShape)
224                 return false;
225             pShape->load(it);
226             _shapes.push_back(pShape);
227             ++it;
228         }
229         return true;
230     }
```

Slika 78: Model's `loadXml` Implementation

- `RoundedRect::load` loads its own attributes and calls the base class `Rect` to do the same.

```
42 void RoundedRect::load(const xml::FileParser::node_iterator& it)
43 {
44     it.getAttribValue("r", _radius);
45     Rect::load(it);
46 }
```

Slika 79: RoundedRect load method

- `Rect::load` loads its own attributes and calls the base class `Shape` to do the same.

```
42 void RoundedRect::load(const xml::FileParser::node_iterator& it)
43 {
44     it.getAttribValue("r", _radius);
45     Rect::load(it);
46 }
```

Slika 80: Rectangle load method

- `Shape::load` loads local attributes.

```
44 void Shape::load(const xml::FileParser::node_iterator& it)
45 {
46     it.getAttribValue("fillColor", _fillColor);
47     it.getAttribValue("lineColor", _lineColor);
48     it.getAttribValue("linePattern", _linePattern);
49 }
```

Slika 81: Shape load method

- `Circ::load` loads local attributes and calls the base class `Shape` to do the same.

```
44 void Circ::load(const xml::FileParser::node_iterator& it)
45 {
46     double x=0, y=0, r=0;
47     it.getAttribValue("x", x);
48     it.getAttribValue("y", y);
49     it.getAttribValue("r", r);
50     _circle.center.x = x; _circle.center.y = y; _circle.r = r;
51     Shape::load(it);
52 }
```

Slika 82: Circle load method

3 Databases

3.1 Introduction to Databases

In the previous lectures, various file types were used for I/O operations (text, binary, XML). We observed that we always load the entire file, and when we change something in the model, the writing process starts over "from the beginning."

Some of the disadvantages of this approach are:

- There is no straightforward way to sort and filter data.
- It's not possible to work with a small subset of the data.
- There is no simple way to use data from different files.
- Data is often written multiple times (redundancy).
- There is no access control when other applications depend on our results.
- Data exchange with other applications running in parallel is problematic.

3.2 Introduction to SQL

It can be concluded that in our previous work, besides focusing on the primary goal (processing desired data), a significant focus was on how to manipulate the process of loading and saving data.

These problems were detected very early on, leading to the idea of creating a type of language that falls into the category of a query language.

- **Structured Query Language (SQL)** is the most popular language for processing structured data.
  - It is used for managing relational databases such as MySQL, ORACLE, SQL Server, PostgreSQL, etc.
- The advantage of SQL is that we do not have to specify how to get data from the database. Instead, we simply specify what needs to be retrieved, and SQL does the rest.
- Data is usually organized into logical units called databases.
- Databases are hosted on a server, which can have multiple databases.
- At any given moment, it is possible to have multiple clients processing data on the server within the same database.
  - Clients send requests, and the server processes them, ensuring data consistency.
  - Multiple requests from a single client can be grouped into a transaction that can be committed or rolled back.
- This mode of operation is called the transactional or client-server approach.

3.3 A Brief History of SQL

- The SQL programming language was developed in the 1970s by IBM researchers Raymond Boyce and Donald Chamberlin.
- The language, then known as SEQUEL, was created following the work of Edgar Frank Codd, "A Relational Model of Data for Large Shared Data Banks," in 1970.
- In his work, Codd proposed that all data in a database be represented in relations.
- Based on this theory, Boyce and Chamberlin developed SQL.
- However, it was only a few years later that the SQL language became publicly available.
- In 1979, a company called Relational Software, which later became Oracle, commercially released its own version of SQL, named Oracle V2.
- Since then, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) have considered SQL the standard language for relational databases.
- While major SQL vendors modify the language according to their own needs, most base their SQL programs on the version approved by ANSI.

3.4 History of SQL Standards

- **1986** - ANSI standard SQL.
- **1992** - a.k.a. SQL2: improved integrity.
- **1999** - a.k.a. SQL3: added searching via regular expressions, recursive queries, triggers, support for procedures and flow control, support for non-scalar types...
- **2003** - SQL:2003: added support for XML.
- **2006** - SQL:2006: improvements related to XML.
- **2008** - SQL:2008: improved ORDER BY, added TRUNCATE and FETCH.
- **2011** - SQL:2011: temporary tables.
- **2016** - SQL:2016: polymorphic tables, JSON (Web).
- **2019** - SQL:2019: multidimensional tables.

3.5 Advantages of SQL

Although there are some drawbacks to SQL, such as a clunky interface and inefficiency due to the need for expression parsing, its advantages usually outweigh its disadvantages. SQL is available on various platforms, and its ease of use can help anyone become proficient.

Advantages:

- **Platform Independent (Portable):** SQL can be used on computers with different operating systems and architectures.

- **Fast Query Processing:** Regardless of how large the data is, SQL can retrieve it quickly and efficiently. It can also perform processes like inserting, deleting, and manipulating data relatively quickly.
- **No Special Programming Skills Required:** SQL only requires the use of simple keywords such as "select", "insert into", "update", and "delete". More on this later.
- **Standardized Language:** The standardized language used in SQL makes it very accessible to all users. SQL provides a unified platform and primarily uses English words and statements, making it easy to learn and write, even for those without prior experience.
- **Multiple Views:** Using SQL allows for the creation of multiple different views of the same data, giving different users different perspectives on the structure and content of the database.
- **Open-Source Availability:** Servers developed with an open-source paradigm, such as MariaDB and PostgreSQL, offer free SQL databases that users can utilize. Many server manufacturers also provide free versions with limitations on the number of connections and parallel processes (e.g., Oracle 12c, MS SQL Server 2019 Developer Edition).
- **Interactive:** Many data processing systems, as well as those available via the Web, use databases (and thus SQL) for storing and processing data.

3.6 Databases and SQL: RDBMS and Servers

Currently, the most significant producers of servers for RDBMS (Relational Database Management Systems) are:

- **Oracle:** Oracle, MySQL
- **Microsoft:** MS SQL Server
- **IBM:** IBM DB2
- **Open Source:** PostgreSQL, MariaDB

In addition to these, there are also so-called **in-process** SQL servers that do not require the installation of a separate SQL server to efficiently leverage the advantages offered by SQL.

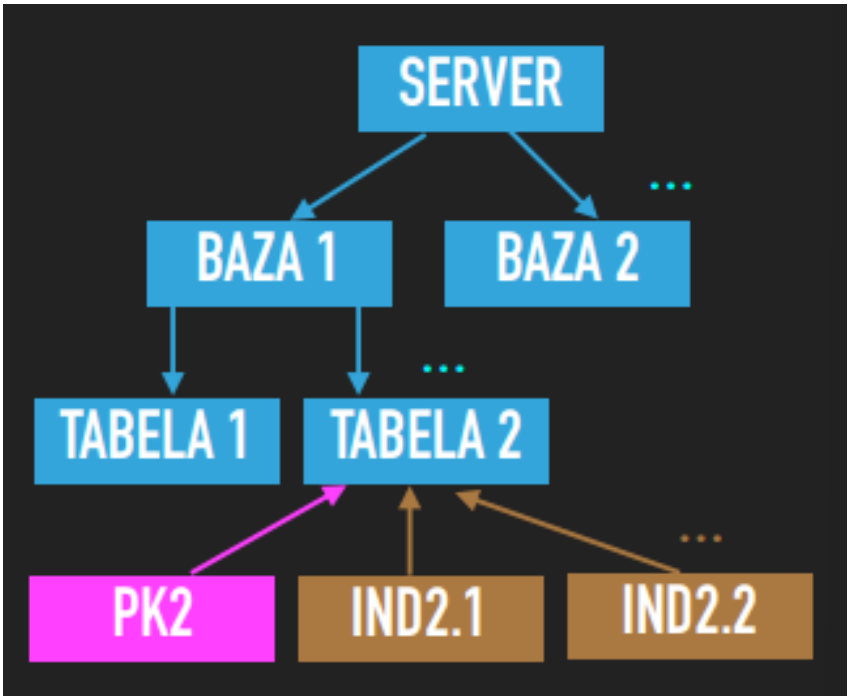
- **SQLite** is the most prominent example of this type of SQL interaction with data.

It is important to emphasize that individual producers (vendors) of RDBMS systems introduce their own extensions, and there are certain differences in the definition of data types.

Slika 83: A selection of prominent and well-known users of SQLite.

3.7 Database and Table Structure

- A single server can have multiple databases.
- A database has one or more tables where data is stored.
  - A table can be viewed as a binary file into which data is entered, organized in structures.
  - The elements of the structure are the columns of the table.
- Each table should have a **primary key**, which is used to uniquely identify a row (record) or a single instance of the data structure.
  - Only one primary key can exist for a single table.
  - The elements of a primary key consist of one or more columns of the table.
- If a primary key is used as a reference from another table, it is called a **foreign key**.
  - In this case, the user cannot delete data defined by the primary key as long as there is at least one record with that foreign key.
  - This is an important principle for maintaining database integrity.
- For faster data filtering and sorting, **indexes** can be used.
  - Indexes can be normal or UNIQUE and are composed of one or more table columns.
- In addition, modern servers provide the ability to use built-in stored procedures, views, triggers, and more.



Slika 84: Hierarchical structure: Server contains Databases, which contain Tables.

3.8 DDL and DML

Regardless of the differences in specific implementations, they all define the necessary components for creating a database, manipulating data, and handling transactions (groups of data manipulations). In this sense, SQL defines two parts of the language:

- **Data Definition Language (DDL)** defines the commands needed to create a database.
  - The most important DDL commands are:
    - **CREATE** - creates an object (table, index, etc.) in the database.
    - **ALTER** - modifies the structure of an existing object in various ways—for example, by adding a column to an existing table.
    - **DROP** - deletes an object from the database, usually irreversibly.
  - For practical work, detailed knowledge of DDL is not required. Graphical tools can be used to generate the database for any existing version of an RDBMS.
- **Data Manipulation Language (DML)** is a subset of SQL used for adding, updating, and deleting data.
  - For application development using SQL, knowledge of only 4 commands is necessary.
  - The acronym **CRUD** (Create, Read, Update, Delete) refers to all the main functions of SQL.
    - The **INSERT INTO** command belongs to the Create group as it is used to add new data.
    - The **SELECT** command belongs to the Read group as it is used to read existing data.
    - The **UPDATE** command belongs to the Update group as it is used to modify (update) existing data.
    - The **DELETE** command belongs to the Delete group as it is used to delete existing data.

3.9 Data Definition Language (DDL) in SQLite

Detailed knowledge of DDL commands is not essential. In practice, the database is very often designed using one of the available graphical tools. For SQLite, there are many free graphical tools available.

Creating tables is done as follows:

```
CREATE TABLE [IF NOT EXISTS] tableName (
    column1 dataType PRIMARY KEY,
    column2 dataType NOT NULL,
    ....
    columnN dataType DEFAULT value,
    optionalConstraintsAndRelations);
```

- **IF NOT EXISTS** is an optional part - the table is created only if it has not been created before.
- The desired table name is entered in the `tableName` field.
- A table can have an arbitrary number of columns, named from `column1` to `columnN`.
- Each column must have a name and a data type (`dataType`).
- The **PRIMARY KEY** can be composed of one or more columns.
- When inserting (saving) data into the table, it is not necessary to save a value for a column if it is declared as nullable (no `NOT NULL` constraint). If it is declared as **NOT NULL**, a value must be provided for that column during insertion.
- A field can have a **DEFAULT** value.

3.9.1 Data Types in SQLite

- **INTEGER:** A signed integer value, stored in 1, 2, 4, or 8 bytes, depending on the magnitude of the value.
  - The following C++ types can be written into columns with this type: `td::BYTE, td::WORD, td::INT2, td::(U)INT4, td::L(U)INT8, td::Decimal, td::Date, td::Time, td::ColorID`, ...
- **REAL:** A floating-point value.
  - `double` and `float` numbers can be stored in columns with this type.
- **TEXT:** A text string value, stored using the database's UNICODE encoding (UTF-8, UTF-16BE, or UTF-16LE).
  - Only `td::String` values will be written to columns of this type.
- **BLOB:** Used for inserting images or other "raw" data.

3.9.2 DDL Example

For demonstration purposes, let's assume we want to create a system for tracking issued invoices via an online platform. For this purpose, it is necessary to create two tables:

- The **Clients** table contains information about customers.

```
CREATE TABLE IF NOT EXISTS Clients (
    ID INTEGER PRIMARY KEY,
    Name TEXT NOT NULL,
    Address TEXT NOT NULL);
```

- The **Invoices** table contains information about issued invoices, including the value of the goods sold and information about the customer.

```
CREATE TABLE IF NOT EXISTS Invoices (
    Number INTEGER PRIMARY KEY,
    Date INTEGER NOT NULL,
    ClientID INTEGER NOT NULL,
    Amount INTEGER NOT NULL,
    FOREIGN KEY (ClientID)
        REFERENCES Clients (ID));
```

  To ensure database consistency, a foreign key is added to the `Invoices` table, which ensures that the `ClientID` must exist in the `Clients` table in its `ID` column.

3.9.3 Creating an Index

With this, the database is created and ready for data manipulation (insertion, modification, deletion, reading). Sometimes, however, it is desirable to perform searches based on different criteria. Let's say we want to quickly retrieve all invoices but only for one customer.

To make the system search data faster, it is desirable to create an index on the required fields (one or more). This is achieved with the `CREATE INDEX` command, whose syntax in the general case looks like this:

```
CREATE [UNIQUE] INDEX indexName
ON tableName (listOfColumnNames);
```

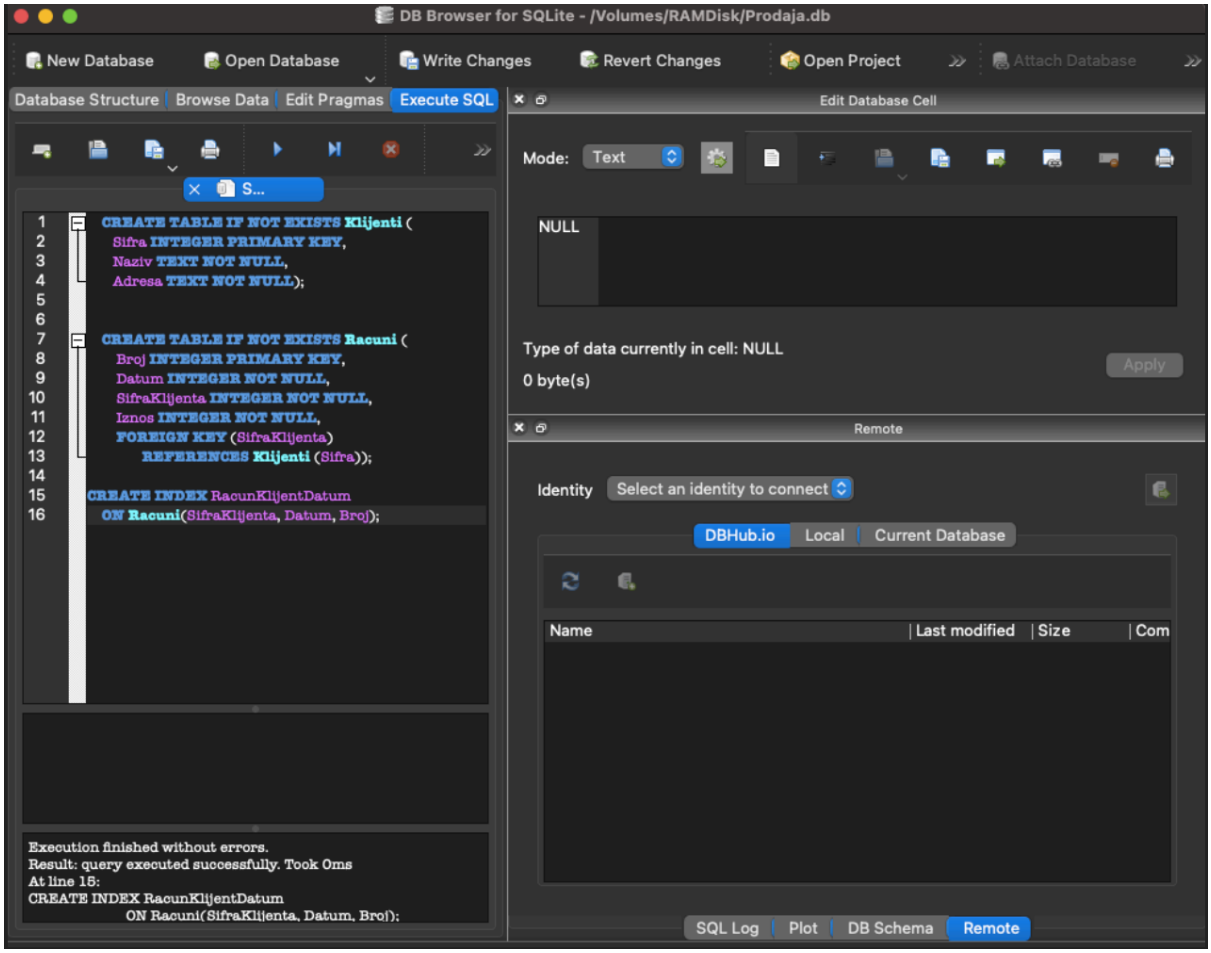
And in our case:

```
CREATE INDEX InvoiceClientDate
ON Invoices(ClientID, Date, Number);
```

3.9.4 Using a GUI for DDL

As mentioned earlier, modifying the database structure is easiest to do through a SQLite IDE.

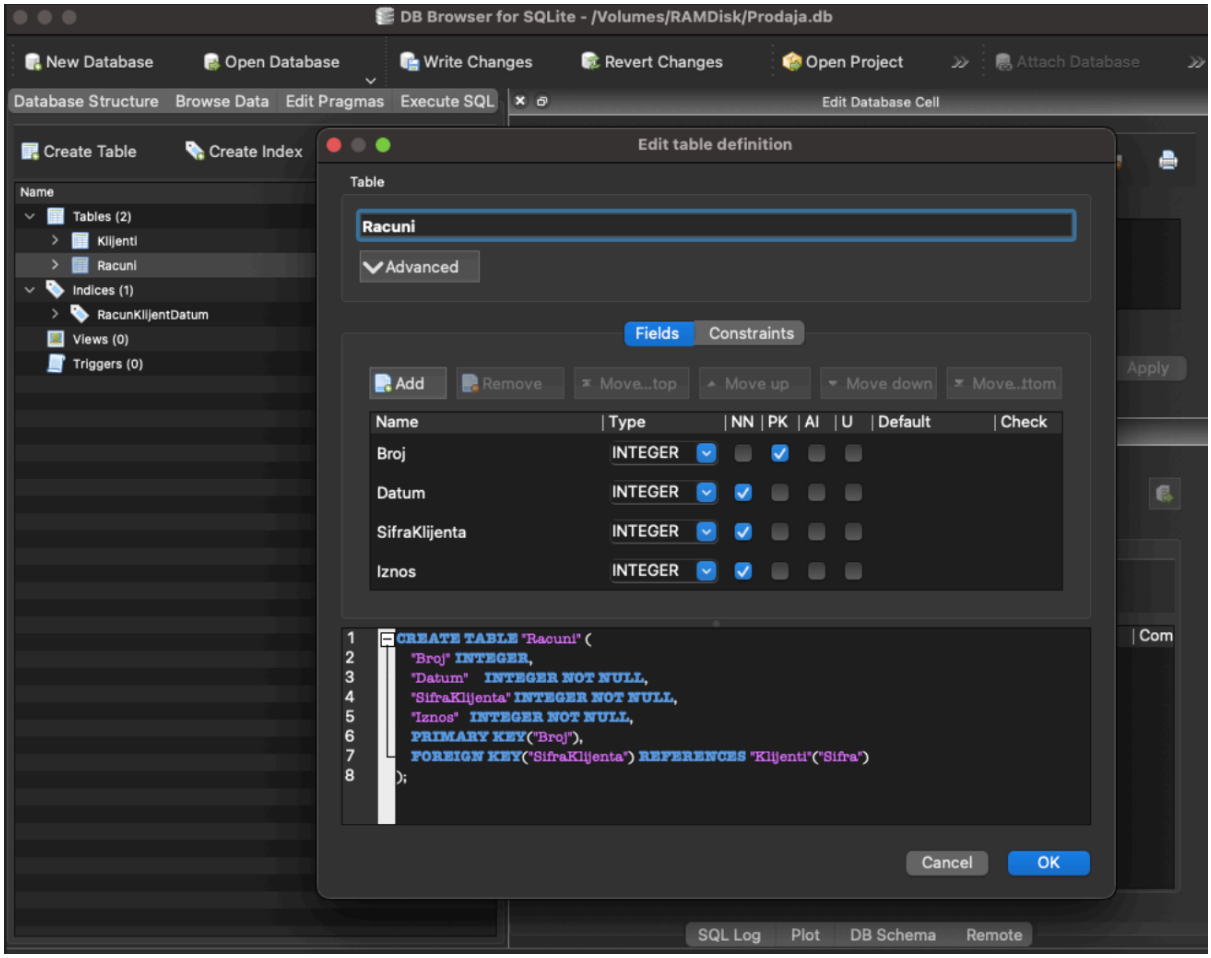
- **DB Browser for SQLite** is free and available for Windows, macOS, and Linux: *https://sqlitebrowser.org/dl/*
- A new database can be created by clicking on **New Database**.
- After that, the structure can be created by using a previously prepared script (image on the right) or by graphically editing the tables (next slide).
- In addition to DDL, DB Browser for SQLite also allows for data manipulation.



Slika 85: Creating a database schema by running a DDL script in the "Execute SQL" tab of DB Browser for SQLite.

Graphical entry (modification) of tables can be done by using the **Create Table** button if the desired table has not already been created, or by right-clicking on an existing table and selecting **Modify** from the menu. After that, you can add/change the names, types, and constraints for the columns.

- Use the **Fields** tab for columns, as shown in the image on the right.
- Use the **Constraints** tab, as shown in the image on the right.
- For practice, create several tables using DB Browser for SQLite.



Slika 86: Using the graphical interface in DB Browser for SQLite to define or modify a table's structure.

3.10 Data Manipulation Language (DML) - INSERT

Data is inserted into the database (table) using the `INSERT INTO` command.

```
INSERT INTO tableName (column1, column2,..)
VALUES (value1, value2,...);
```

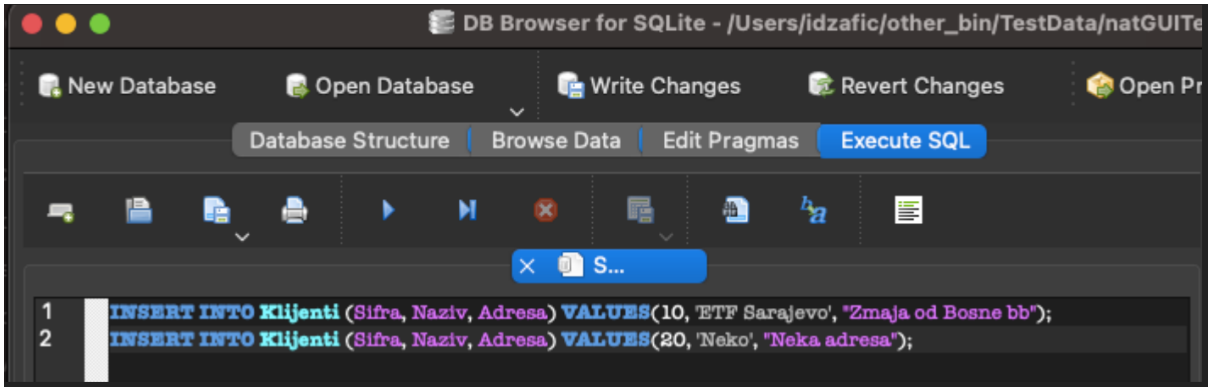
- `value1` is inserted into `column1`, `value2` into `column2`, and so on.

Example 1: If we want to enter information about two clients into the `Clients` table:

```
INSERT INTO Clients (ID, Name, Address)
VALUES(10, 'ETF Sarajevo', "Zmaja od Bosne bb");

INSERT INTO Clients (ID, Name, Address)
VALUES (20, 'Neko', "Neka adresa");
```

- With this, the `Clients` table would contain two records.

The execution of SQL commands is ultimately done in C++. For practice, you can try it via the **Execute SQL** button in DB Browser for SQLite.



Slika 87: Executing INSERT statements in DB Browser for SQLite.

Example 2: Enter information for several invoices for clients 10 and 20:

```
INSERT INTO Invoices (Number, Date, ClientID, Amount) VALUES(1, 0, 10, 1000);
INSERT INTO Invoices (Number, Date, ClientID, Amount) VALUES(2, 0, 20, 2000);
INSERT INTO Invoices (Number, Date, ClientID, Amount) VALUES(3, 0, 20, 3000);
INSERT INTO Invoices (Number, Date, ClientID, Amount) VALUES(4, 0, 10, 4000);
INSERT INTO Invoices (Number, Date, ClientID, Amount) VALUES(5, 0, 10, 5000);
```

- Invoices 1, 4, and 5 are registered to the client with ID 10 (ETF), and invoices 2 and 3 are for the client with ID 20 (Neko).
- For now, we have set the `Date` to zero (more on this in C++).
- An integer value was used for the amount.
  - When we work with C++, we will see that this data is actually of type `td::Decimal2` (two decimal places). More on this later.
- It is important to note that instead of writing the client's name in each record of the `Invoices` table, the integer value of `ClientID` (FOREIGN KEY) is used. This eliminates redundancy and ensures data consistency.
  - If you want to change the client's name, it is changed only in the `Clients` table, and all invoices will then reflect the new (correct client name).

3.11 Reading Data - SELECT

The `SELECT` command is used to read data from the database. If you want to load a specific number of columns from a single table, it can be done with:

```
SELECT column1, column2, ..., columnN FROM tableName
```

For example:

```
SELECT ID, Name, Address from Clients;
```

This would load the following values:

	Sifra	Naziv	Adresa
1	10	ETF Sarajevo	Zmaja od Bosne bb
2	20	Neko	Neka adresa

Slika 88: Result of a simple SELECT query on the Clients table.

3.11.1 Filtering with WHERE

Very often, the `SELECT` command is used to filter data. This is enabled by adding optional conditions using the `WHERE` keyword:

```
SELECT column1, column2, ..., columnN FROM tableName WHERE condition;
```

- In this case, the 'condition' is a set of logical operations on one or more columns.

For example, if we want to display all invoices for the client with ID 10 (ETF) from the `Invoices` table:

```
SELECT Number, Amount from Invoices WHERE ClientID=10;
```

This would load the data:

	Broj	Iznos
1	1	1000
2	4	4000
3	5	5000

Slika 89: Result of a SELECT query with a WHERE clause.

The following operators can be used to form conditions:

- `=` (equality): e.g., `WHERE ClientID = 20`
- `>` (greater than), `<` (less than): e.g., `WHERE Amount > 2000`
- `>=` (greater than or equal to), `<=` (less than or equal to): e.g., `WHERE Amount <= 4000`
- `LIKE` (searching for part of the text): e.g., `WHERE Name LIKE 'ETF%'`
- Logical `AND` and `OR` operators: e.g., `WHERE Amount >= 2000 AND Amount <= 4000`

3.11.2 Sorting with ORDER BY

In addition to filtering data, the `SELECT` command can also be used for sorting data. This is enabled by adding optional conditions using the `ORDER BY` keyword:

```
SELECT column1, column2, ..., columnN
FROM tableName
WHERE condition
ORDER BY listOfColumnNames [ASC | DESC];
```

For example, if we want to sort the invoices of client 10 in descending order (from the largest amount to the smallest):

```
SELECT Number, Amount from Invoices WHERE ClientID=10 ORDER BY Amount DESC;
```

This would load the data in the following way:

	Broj	Iznos
1	5	5000
2	4	4000
3	1	1000

Slika 90: Result of a SELECT query sorted in descending order.

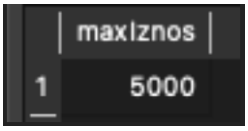
3.11.3 Aggregating Data

SQL is useful not only for selecting or maintaining databases, but also for aggregating data. SQL has numerous useful aggregate functions, including `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`.

Example 1: If we want to find the largest amount of an issued invoice for the client with ID 10:

```
SELECT MAX(Amount) as maxAmount from Invoices WHERE ClientID=10;
```

Which would give the result:



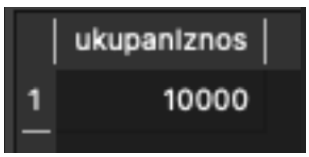
	maxIznos
1	5000

Slika 91: Result of the MAX() aggregate function.

Example 2: If we want to find the total amount of all invoices issued to the client with ID 10:

```
SELECT SUM(Amount) as totalAmount from Invoices WHERE ClientID=10;
```

Which would give the result:



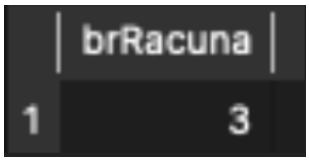
	ukupanIznos
1	10000

Slika 92: Result of the SUM() aggregate function.

Example 3: If we want to know how many invoices were issued to the client with ID 10:

```
SELECT COUNT(*) as numInvoices from Invoices WHERE ClientID=10;
```

Which would give the result:



	brRacuna
1	3

Slika 93: Result of the COUNT() aggregate function.

3.11.4 Grouping and Aggregating with GROUP BY

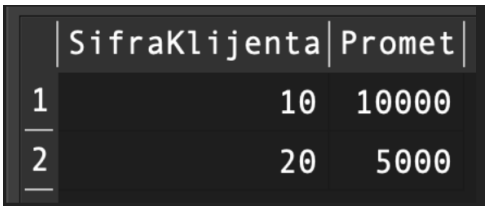
The previously shown aggregation functions can be executed on records grouped by a certain condition defined with `GROUP BY`.

- This allows for the simultaneous aggregation of multiple subsets of data.

For example, if you want to display the total sum of all invoices per client:

```
SELECT ClientID, SUM(Amount) AS Turnover
FROM Invoices
GROUP BY ClientID
```

This would generate the following result:



	SifraKlijenta	Promet
1	10	10000
2	20	5000

Slika 94: Result of using SUM() with GROUP BY.

- `GROUP BY` can contain more than one column. In that case, the names are listed with a comma between them.
- **Note:** The name of each column can be changed using the `AS` keyword, as was done in the aggregation function examples.
- **Note from professor:** Add examples with `LEFT JOIN` and nested selects (`select` in `select`).

3.12 Joining Data from Multiple Tables

It is often necessary to combine data from different tables. For our database with two tables (`Clients`, `Invoices`), it would be interesting to see the names of the clients instead of their IDs. This is a so-called **INNER JOIN**, and it can be implemented with the `WHERE` keyword:

```
SELECT A.Number, A.ClientID, B.Name, B.Address, A.Amount
FROM Invoices A, Clients B
WHERE A.ClientID = B.ID;
```

Which would yield the following result:

	Broj	SifraKlijenta	Naziv	Adresa	Iznos
1	1	10	ETF Sarajevo	Zmaja od Bosne bb	1000
2	2	20	Neko	Neka adresa	2000
3	3	20	Neko	Neka adresa	3000
4	4	10	ETF Sarajevo	Zmaja od Bosne bb	4000
5	5	10	ETF Sarajevo	Zmaja od Bosne bb	5000

Slika 95: Result of an INNER JOIN between the Invoices and Clients tables.

In this case, previously mentioned filtering conditions can be added within the `WHERE` clause.

3.12.1 Joining a Table with a Subquery

Data can also be joined using nested queries (`SELECT` within a `SELECT`). For example, if you want to display the turnover for all customers, and also show the customer names, you need to create the following SQL command:

```
SELECT K.ID, K.Name, S.Turnover
FROM Clients K, (SELECT ClientID, SUM(Amount) AS Turnover
                 FROM Invoices
                 GROUP BY ClientID) S
WHERE K.ID = S.ClientID
```

Executing this command would generate the following result:

	Sifra	Naziv	Promet
1	10	ETF Safajevo	10000
2	20	Neko	5000

Slika 96: Result of joining a table with a subquery.

- In this case, the data from table K and table S are joined by their ID (`K.ID = S.ClientID`).
- Table S in this case is the result of the second select statement.

3.12.2 Joining with LEFT JOIN

In the previous examples, a so-called **INNER JOIN** was used, where data that exists in both tables (or in all tables being joined) is combined.

- In certain cases, we want to take all data from the first table and replace missing data from the second table with a default value.

- This type of join can be performed using **LEFT JOIN**.
- To demonstrate `LEFT JOIN`, let's add one more Client:

```
INSERT INTO Clients (ID, Name, Address)
VALUES (30, 'Client without invoices', 'Address not for shopping');
```

- If we were to execute the query from the previous slide, we would get the same result as before.
  - The client with ID 30 is only in the `Clients` table, and thus is not in the set of common data (it is not in both tables).
  - Client 30 is not displayed in the result.

If we want to have the client with ID 30 in the results, and show their turnover as zero, it is necessary to execute the command that joins the two tables using a `LEFT JOIN`:

```
SELECT K.ID, K.Name, IFNULL(S.Turnover, 0) as Turnover
FROM Clients K
LEFT JOIN (SELECT ClientID, SUM(Amount) AS Turnover
           FROM Invoices GROUP BY ClientID) S
ON K.ID = S.ClientID
```

- `LEFT JOIN` must be combined with `ON`, which defines the join criterion.
- The `IFNULL` function checks if the first parameter is null. If it is, it replaces it with the second parameter. If not, it displays the first parameter.

The previous example gives the following result:

	Sifra	Naziv	Promet
1	10	ETF Safajevo	10000
2	20	Neko	5000
3	30	Klijent bez racuna	0

Slika 97: Result of a LEFT JOIN, showing clients with no matching records.

3.13 Modifying Data - UPDATE

To change existing data, the `UPDATE` command is used:

```
UPDATE tableName
SET column1=value1, column2=value2,..., columnN=valueN
WHERE condition;
```

Example 1: If you want to change the amount of invoice number 2 to 2500:

```
UPDATE Invoices
SET Amount=2500
WHERE Number=2;
```

Example 2: If you want to change the name and address of the client with ID 20:

```
UPDATE Clients
SET Name='ASA Group', Address='Dzemala Bijedica 110'
WHERE ID=20;
```

With this, the content of all invoices would be (using the `SELECT` from the previous slide):

Broj	SifraKlijenta	Naziv	Adresa	Iznos
1	10	ETF Sarajevo	Zmaja od Bosne bb	1000
2	20	ASA Grupacija	Dzemala Bijedica 110	2500
3	20	ASA Grupacija	Dzemala Bijedica 110	3000
4	10	ETF Sarajevo	Zmaja od Bosne bb	4000
5	10	ETF Sarajevo	Zmaja od Bosne bb	5000

Slika 98: Data after executing UPDATE statements.

3.14 Deleting Data - DELETE

To delete existing data, the `DELETE` command is used:

```
DELETE FROM tableName WHERE condition;
```

- **Note:** If the 'condition' is omitted, the entire content of the 'tableName' table will be deleted.

Example 1: If we want to delete invoice number 5:

```
DELETE FROM Invoices WHERE Number=5;
```

With this, the content of all invoices (SELECT command taken from two slides ago) would be:

Broj	SifraKlijenta	Naziv	Adresa	Iznos
1	10	ETF Sarajevo	Zmaja od Bosne bb	1000
2	20	ASA Grupacija	Dzemala Bijedica 110	2500
3	20	ASA Grupacija	Dzemala Bijedica 110	3000
4	10	ETF Sarajevo	Zmaja od Bosne bb	4000

Slika 99: Data after deleting a record.

Q1: What would happen if the following command were executed on the existing data?

```
DELETE FROM Clients WHERE ID=20;
```

A1: An error would occur, and nothing would be deleted. **Q2: Why?**

- Because the RDBMS takes care of data consistency.
- Since the `Invoices` table has records that use a `FOREIGN KEY` with the value 20, deletion will not be allowed until all records in the `Invoices` table that have a `ClientID` = 20 are deleted.

4 Databases and C++ (natID Approach)

4.1 The IDatabase Interface

- The natID framework provides an identical interface for accessing data from SQLite databases, professional servers (ORACLE, MSSQL, IBM DB2,...), and open-source servers (PostgreSQL, MariaDB,...).
- This functionality is enabled by including the dataProvider library.
- To access a database, it is first necessary to create a dp::IDatabase interface, which is achieved using the dp::create method exported by the dataProvider library (header <dp/IDatabase.h>).

```
IDatabase* create(IDatabase::ConnType connType,
                  IDatabase::ServerType serverType);
```

  - The first parameter specifies the connection type and can have one of the following values: CT_ODBC, CT_CLI, CT_OCI, CT_PG, CT_SQLITE.
    - In our case, it is always CT_SQLITE.
  - The second parameter specifies the server type and can have one of the following values: SER_MSSQL = 1, SER_ORACLE, SER_DB2, SER_FIREBIRD, SER_POSTGRE, SER_MYSQL, SER_SYBASE, SER_TIMESTEN, SER_SQLITE3, SER_SQLITE_4.
    - In our case, it is always SER_SQLITE_3.
- After creating the interface, you need to connect to the desired database using the connect method, which is part of the IDatabase interface:

```
bool connect(const td::String& connectionString,
             const td::String* pUserName = 0,
             const td::String* pPassword = 0) = 0;
```

4.1.1 IDatabase and IDatabasePtr

After finishing work with the database, the resources allocated by the dp::IDatabase interface must be released.

- This can be done in two ways:
  - Manually, by calling the release method on the IDatabase interface instance.
  - Or automatically, if the IDatabase pointer is placed into an IDatabasePtr object.
    - In this case, the destructor of IDatabasePtr calls the release method.
    - **The automatic method is recommended** and is used in all examples in these lectures.
- It should be emphasized that in most cases of working with different servers, only the connection string differs, while everything else remains identical.

4.2 The IStatement Interface

- After a successful connection to the database (and resolving the resource release issue), you can start manipulating the database data using SQL (DML).

- All commands to the server are executed using the `dp::IStatement` interface (located in the `<dp/IStatement.h>` header).
- An instance of the `IStatement` interface can be created by calling the `createStatement` method on an `IDatabase` interface instance:

```
virtual IStatement* createStatement(
    const td::String& strStatement) = 0;
```

  - The `strStatement` parameter is a string containing one of the four possible commands (SELECT, INSERT, UPDATE, DELETE).
    - Since an `IStatement` instance can be called multiple times (e.g., we can insert multiple records with one INSERT statement), this parameter uses a question mark ('?') as a parameter marker.
      - Some servers allow calls to so-called stored procedures, so these parameters can be both input and output.
      - In the case of SQLite, it is only possible to use input parameters.
    - To "bind" the parameters to the infrastructure, it is necessary to specify where the values of individual parameters are taken from (more on this in the examples) via local variables of the appropriate type.
- After binding the parameters, the command is executed by calling the `execute` method on the `IStatement` instance:

```
virtual bool execute() = 0;
```

  - Returns `true` if the command was successfully executed on the server.

4.3 Pseudo-algorithm for INSERT, DELETE, UPDATE

Let's assume we have created the `IDatabase` interface and connected to the database:

```
dp::IDatabasePtr pDB(dp::create(
    dp::IDatabase::ConnType::CT_SQLITE,
    dp::IDatabase::ServerType::SER_SQLITE3));

if (!pDB->connect("R/mySQLiteDB.db"))
    return false;
```

- Since INSERT/DELETE/UPDATE commands modify data on the server, they depend on the accuracy of the data sent by the user.
- To enable data consistency to be maintained, transactions must be used.
- For this purpose, `dp::Transaction` is used (header `<db/IDatabase.h>`), which requests an `IDatabase` instance in its constructor.
  - If everything is OK, all INSERT, DELETE, UPDATE commands will be permanently activated after a `commit` call on an instance of the `db::Transaction` class.
  - If something was not OK, `rollBack` returns the database state to the moment before the `dp::Transaction` was started.

- The destructor of a `db::Transaction` instance automatically performs a `rollBack` if `commit` was not previously called.

4.3.1 Step-by-Step Execution

Let's assume we want to insert data using the INSERT command (the process is similar for other commands). Inserting/deleting/updating data is done in the following 7 steps:

1. Create an `IStatement` with an instantiated and successfully connected `pDB`:

```
dp::IStatementPtr pStat(pDB->createStatement(
   "INSERT INTO TableName (COL1,COL2,COL3) VALUES(?,?,?)"));
```

2. Define local variables for the parameters (there must be as many local variables as there are question marks to bind):

```
DataType1 localVar1;
DataType2 localVar2;
DataType3 localVar3;
```

3. Bind the SQL command parameters (`IStatement`) with local variables of the appropriate type:

```
dp::Params par(pStat->allocParams());
par << localVar1 << localVar2 << localVar3;
```

4. Start the transaction:

```
dp::Transaction tr(pDB);
```

5. Set the data from a source into the local variables:

```
localVar1 = getPreparedData(i, 0); // get value of first var
localVar2 = getPreparedData(i, 1); // get value of second var
localVar3 = getPreparedData(i, 2); // get value of third var
```

6. Call the `execute` method, which copies the values of the local variables and sends them to the server (the SQL command is completed):

```
if (!pStat->execute()) // execute takes local var values as params
{
    tr.rollBack(); // first thing to do if something is not ok
    td::String strErr;
    pStat->getErrorStr(strErr); // get error description
    std::cout << "Error! " << strErr << td::endl; // print error
    return false;
}
```

   In case the `execute` method returns false, it is necessary to immediately perform a `rollBack` (to release locks on the database so that other users can start processing data sooner).

   - Details of the error can be obtained by calling `getErrorStr` on the `IStatement` instance.

7. Go to step 5 if there is more data; otherwise:

```
tr.commit();
return true;
```

4.3.2 Complete Code Example

```
bool pseudoInsert(dp::IDatabasePtr& pDB,
    const cnt::SafeFullVector<SomeStructure>& data)
{
    // create statement
    dp::IStatementPtr pStat(pDB->createStatement(
        "INSERT INTO TableName (COL1,COL2,COL3) VALUES(?,?,?)"));

    // declare local variables to which data will be transferred
    DataType1 localVar1;
    DataType2 localVar2;
    DataType3 localVar3;

    // bind local variables with parameters
    dp::Params par(pStat->allocParams());
    par << localVar1 << localVar2 << localVar3;

    dp::Transaction tr(pDB); // start transaction
    size_t nRecordsToTransfer = data.size();
    for (size_t i=0; i<nRecordsToTransfer; ++i)
    {
        // set values of local variables
        localVar1 = data[i].valCol1; // get value of first var
        localVar2 = data[i].valCol2; // get value of second var
        localVar3 = data[i].valCol3; // get value of third var

        if (!pStat->execute()) // execute uses local var values
        {
            tr.rollBack(); // first thing to do if something is not ok
            td::String strErr;
            pStat->getErrorStr(strErr); // get error description
            std::cout << "Error! " << strErr << td::endl; // print error
            return false;
        }
    }
    tr.commit();
    return true;
}
```

4.4 The `IStatement` Interface for SELECT

- Unlike INSERT, UPDATE, and DELETE commands, the SELECT command can have input parameters and usually returns one or more records, which can have one or more columns.
  - Parameters are bound in the same way as previously described and is identical for all command types.
  - To retrieve the values of the data set (array of structures), each column to be retrieved (not necessarily all) must be bound to a local variable of the appropriate type.
    - For this purpose, column binding is performed, where the name of the requested column that SELECT returns and the local variable (of the appropriate type) where the column's value will be stored are sent to the `IStatement` instance.
- If `execute` was successful, one must then iterate through all the returned records (rows) that the server sends as a result of executing the SELECT command.

– This is achieved using the `moveNext` method on the `IStatement` interface (only in the case of SELECT).

```
virtual bool moveNext() = 0;

```

– It returns `true` if data transfer from the `IStatement` interface to the local variables was successful.

- The SELECT command does not require the use of `db::Transaction` because it does not modify values on the server (SELECT cannot violate data consistency).

4.5 Pseudo-algorithm for SELECT

Loading selected data can be done in the following 5 steps:

1. Create a SELECT `IStatement` with an instantiated and successfully connected `pDB`:

```
dp::IStatementPtr pStat(pDB->createStatement(
    "SELECT COL1, COL2 FROM TableName WHERE COL3=?"));
// also possible if COL1 and COL2 are in the returned set
// (the named column variant is better)
// dp::IStatementPtr pStat(pDB->createStatement(
//    "SELECT * FROM TableName WHERE COL3=?"));
```

2. Bind the input parameters with local variables:

```
// bind local variables with parameters
DataType1 localVar1 = param1; // parameter is set only once
dp::Params par(pStat->allocParams());
par << localVar1;
```

3. Bind the column names of the resulting set with local variables:

```
DataType2 localCol1;
DataType3 localCol2;
dp::Columns cols(pStat->allocBindColumns(2));
cols << "COL1" << localCol1 << "COL2" << localCol2;
```

4. Execute the command using `execute` (sends parameters to the server, and if OK, the server returns the resulting data set):

```
if (!pStat->execute()) // only once
    return false;
```

5. Using `moveNext`, iterate through the resulting set:

```
while (pStat->moveNext()) // iterate through the returned data set
{
    // set the values of local variables into a return vector
    SomeStructure dataStructure;
    dataStructure.valCol1 = localCol1;
    dataStructure.valCol2 = localCol2;
    data.push_back(dataStructure);
}
```

With this, all the data that the server sent as a result of the SELECT query has been loaded.

4.5.1 Complete Code Example

```
bool pseudoSelect(dp::IDatabasePtr& pDB,
    cnt::PushBackVector<SomeStructure>& data, DataType1 par1)
{
    // create SELECT statement
    dp::IStatementPtr pStat(pDB->createStatement(
        "SELECT COL1, COL2 FROM TableName WHERE COL3=?"));

    // declare local variables for parameters and results
    DataType1 localVar1 = par1; // parameter is set only once
    DataType2 localCol1;
    DataType3 localCol2;

    // bind local variables with parameters
    dp::Params par(pStat->allocParams());
    par << localVar1;

    // bind local variables with column names
    dp::Columns cols(pStat->allocBindColumns(2));
    cols << "COL1" << localCol1 << "KOL2" << localCol2;

    if (!pStat->execute())
        return false;

    while (pStat->moveNext()) // iterate through returned data set
    {
        // set values of local variables into the return vector
        SomeStructure dataStructure;
        dataStructure.valCol1 = localCol1;
        dataStructure.valCol2 = localCol2;
        data.push_back(dataStructure);
    }
    return true;
}
```

4.6 The `IDataSet` Interface

- `IDataSet` collects all data for the requested SELECT query and releases the lock on the server.
- It has the ability to export to XML.
- It works much faster than collecting data in a `vector`/`PushBackVector` or some other container.
  - It has much better memory management.
  - It cannot cause a deadlock on the server.
- It is used in GUI interactions with databases.
- It is used for creating reports.

5 Graphical User Interface and C++ (natID Approach)

5.1 Introduction to natGUI

- The `natID` framework provides the necessary infrastructure for developing graphical applications using the C++ programming language.
  - `natGUI` dynamic library (`gui` namespace).
- Modern desktop graphical applications have a simple user interface that includes:
  - Various icons and symbols used to achieve better visualization.
  - The ability to choose the language used in the application's user interface.
  - The ability to store application attributes (settings) in the OS, so that after restarting, the application retains the desired properties.
    - Windows: registry
    - macOS: plist
    - Linux: XML property schemas
  - A menu through which the user quickly and easily "communicates" with the application modules.
  - A toolbar that should provide the most frequently used application modules.
- Common to all operating systems is the use of the term "Window" for the main container of all graphical elements of the user interface.

5.2 Creating a Graphical Application

To create any graphical application using the `natID` framework, it is necessary to implement two classes:

- The first class must be inherited from `gui::Application` and represents the entry point into the application.
  - There can be only one instance of this class, and it is placed in the `main` routine.
  - This class must implement the virtual method `createInitialWindow`, which opens the initial window (form) where the other elements will be located.
  - This is also the second class that must be "overloaded" from the `natID` framework.
- The second class must be inherited from `gui::Window` and represents the first visual element that will appear after the application starts.
- The instance of the class inherited from `gui::Application` is located in the `main` routine.
- The instance of the class inherited from `gui::Window` is stored within `gui::Application`.
  - The user does not need to delete it (`natID` takes care of this).

5.3 Example of a Minimal GUI Application

A demonstration of a minimal GUI application that does nothing but display a window that can be maximized/minimized and closed is provided in the attachment `01_AppWnd` of this lecture.

The `appWnd.cmake` contains almost all the instructions as most modules with executable code, with the addition of using the `natGUI` dynamic library, which was achieved with the following line in `appWnd.cmake`:

```
target_link_libraries(${APPWND_NAME}
    debug ${MU_LIB_DEBUG} debug ${NATGUI_LIB_DEBUG}
    optimized ${MU_LIB_RELEASE} optimized ${NATGUI_LIB_RELEASE})
```

In addition, for macOS users, it is required to specify the basic application parameters via an XML (plist) file, which is achieved with:

```
setTargetPropertiesForGUIApp(${APPWND_NAME} ${APPWND_PLIST})
```

5.3.1 The `main.cpp` Entry Point

The entry point is the `main.cpp` file, whose content is:

```
#include "Application.h"
#include <gui/WinMain.h>

int main(int argc, const char * argv[])
{
    Application app(argc, argv);
    app.init("BA");
    return app.run();
}
```

- The `Application` class definition is included only in `main.cpp` and nowhere else.
- The choice of text resources provides multi-language support. In this case, it is necessary to create the resources needed for translation into various languages. The choice of the tag for translation (in this case "BA") is arbitrary, but all translations for this tag must be defined in the resource files.
- As we've already said, the `app` object belongs to the `Application` class, which is actually inherited from `gui::Application` and is located in the header file `Application.h`.
  - The constructor requires passing the input parameters of the `main` function.
    - From these parameters, the framework finds data about the name of the executable file and its location.
  - The `init` method initializes regional settings and translations according to the requested language.
    - In this case, "BA" stands for Bosnian.
  - The `run` method starts the graphical application, which will have the ability to react to all modern graphical gestures (mouse, keyboard, trackpad, ...).

5.3.2 The `Application.h` File

The complete implementation of the `Application` class is provided in the `Application.h` header file, which contains:

```
#include <gui/Application.h>
#include "MainWindow.h"

class Application : public gui::Application
{
protected:
    gui::Window* createInitialWindow() override
    {
        return new MainWindow();
    }
public:
    Application(int argc, const char** argv)
    : gui::Application(argc, argv)
    {
    }
};
```

- The virtual method `createInitialWindow`, declared in `gui::Application`, needs to be implemented in our own application class.
- The constructor initializes `gui::Application` with parameters obtained from the operating system (taken from the `main` function).
- The `Application` constructor forwards parameters to the `gui::Application` constructor.
- The main task of the `Application` class is to "produce" the initial window, which it does within the virtual `createInitialWindow` method (virtually declared within `gui::Application`).

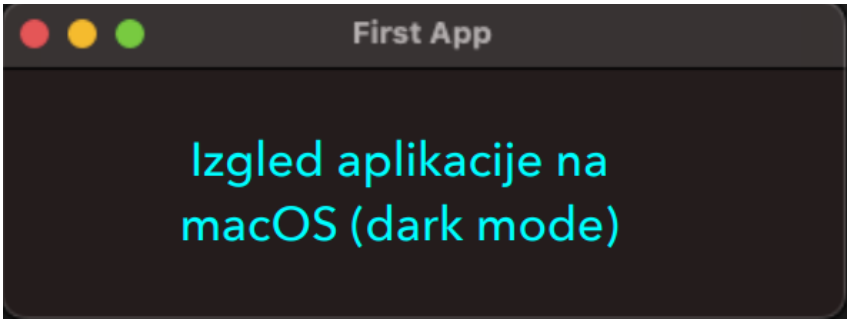
5.3.3 The `MainWindow.h` File

The `MainWindow` class is implemented in the `MainWindow.h` header file, which contains:

```
#include <gui/Window.h>
#include "MenuBar.h"
#include "ToolBar.h"

class MainWindow : public gui::Window
{
protected:
    MenuBar _mainMenuBar;
    ToolBar _toolBar;
public:
    MainWindow()
    : gui::Window(gui::Geometry(50, 50, 1200, 480))
    {
        setTitle(tr("appTitle"));
        _mainMenuBar.setAsMain(this);
        setToolBar(_toolBar);
    }
};
```

- A "window" will be created on the current display at coordinates (50,50) and will have a width of 1200 and a height of 480.

- The `setTitle` method sets the window title.
- The `MainWindow` class extends the `gui::Window` class by defining its dimensions or complete geometry (coordinates of the top-left point as well as width and height).
- This completes everything needed to create the first portable GUI application.
- It does nothing except that the window's dimensions can be changed, and it can be closed.



Slika 100: Appearance of the application on macOS (dark mode).

5.4 Menu and Toolbar

- Modern applications that have a graphical interface for faster communication include a menu and a toolbar.
- Depending on the operating system, the menu can appear in the window (Windows) or be placed in the upper part of the display and be visible for only one menu at a time—the one from the current application (macOS).
- Usually, the window had a special so-called title bar where only the name was displayed, next to three buttons for closing, minimizing, and maximizing the window.
  - The toolbar is usually added below this area (sometimes inside).
- Modern graphical application design utilizes the empty space reserved for the title and includes the toolbar in it.
  - The toolbar usually has certain buttons with symbols (icons), descriptive text, and so-called popup (or popover) elements can also be added.
  - This utilizes the space, and the application becomes clearer and easier to use.
- `natID` uses a modern approach (embedding the toolbar in the title space).

5.5 Graphical Resources

- One of the basic requirements for applications is the ability to work with different languages and alphabets.
- To enable support for multiple languages, all visible texts must be set up so that they can be changed dynamically.

  - The basic idea is to never write the complete text in one of the supported languages directly in the code.
  - Instead, its ID is used for the text, which is then passed to the `tr` method (short for translate), which is part of the framework and belongs to the base class `gui::NatObject`.
  - The text itself that will be displayed is located in a configuration file for a specific language and will be loaded when the application starts.
- Because of this, it is necessary to create translation XML files for each supported language/alphabet.

5.5.1 Icons and Symbols

- Bitmap images (icons) are very often used for the toolbar as well as for other elements of a graphical application.
- In addition to bitmap images, `natID` also supports its own XML-based vector format for symbols.
  - The advantage of vector symbols is that they can be easily adapted to the so-called light or dark color schemes supported by all modern systems.
    * Until a few years ago, it was exclusively a light theme, so all texts were dark.
  - Bitmap images cannot easily change the color of their content in relation to the currently selected OS mode (light or dark).
    * Because of this, it is necessary to choose images that are clearly visible in both cases (they should not be too dark or too light).
- It is very important that the path to the images/symbols is never specified in the code, but that they (similar to texts) are replaced with corresponding IDs, and then the exact path to the image/symbol is specified in the configuration XML file.

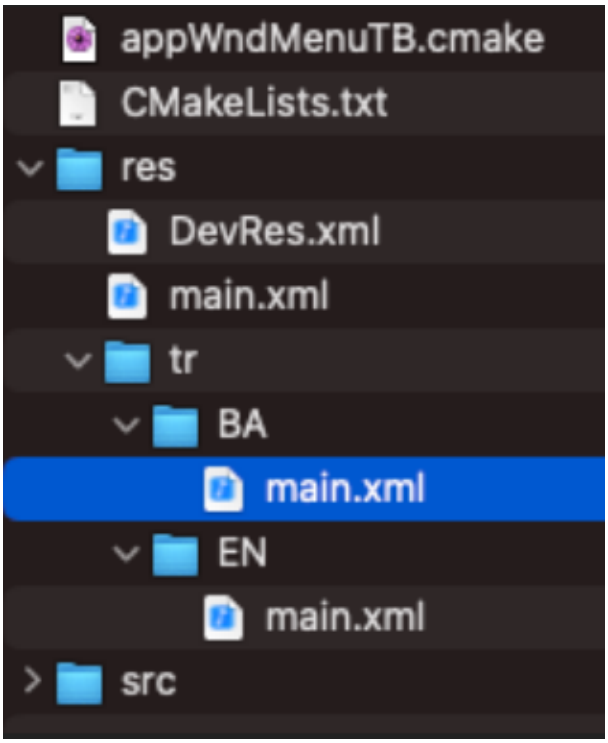
5.6 Defining Resources via XML

- Project `02_AppWndMenuTB`, which is part of the lecture attachments, will be used to demonstrate the configuration of resources (text, png, symbols).
- An application that requires any resource must make the following settings:
  - Create a `res` folder that is at the same level as the `src` folder.
  - Inside this folder, there must be an XML file `DevRes.xml`. For our example, the content of this file is:

```
<?xml version='1.0'?>
<DevRes regionals="Work/Common/Regionals">
    <Langs>
        <Lang id="EN" name="English"/>
        <Lang id="BA" name="Bosanski"/>
    </Langs>
    <ArtWork>
        <Config path="Work/Common/ResConfigs/natGUI.xml"/>
        <Config path="./main.xml"/>
    </ArtWork>
```

```
    <Translation>
        <Config path="Work/Common/ResConfigs/tr/%s/natGUI.xml"/>
        <Config path="./tr/%s/main.xml"/>
    </Translation>
</DevRes>
```

- Supported translations are located in `tr` subdirectories of the same name.
- The names of the configuration files (images) are relative to the HOME location. The first line contains the configuration needed for the `natGUI` dynamic library. The second line belongs to the current application.
- The names of the translation configuration files are relative to the HOME location. The first line contains the translations needed for the `natGUI` dynamic library. The second line contains the translations for the current application.



Slika 101: The project resource folder structure.

5.6.1 Defining Translations via XML

Translations are entered into specified UTF-8 XML files that must have:

- The root node name "DevRes".
- A nested "Translations" node.
- One or more "Res" nodes with translations (the "id" attribute is the identifier used in the C++ code, the "tr" attribute is the translation that will be dynamically loaded).

```
<?xml version='1.0'?>
<DevRes>
```

```
    <Translations>
        <Res id="open" tr="Otvori"/>
        <Res id="openTT" tr="Otvori postojeci model"/>
        <Res id="save" tr="Snimi"/>
        <Res id="saveTT" tr="Snimi postojeci model"/>
        <Res id="rect" tr="Pravougaonik"/>
        <Res id="rectTT" tr="Crtanje pravougaonika"/>
        <Res id="appTitle" tr="Uvod u GUI"/>
        <Res id="mOpen" tr="Aktivirana je opcija za ucitavanje modela!"/>
        <Res id="mSave" tr="Aktivirana je opcija za snimanje modela!"/>
        <Res id="mRect" tr="Aktivirana je opcija za crtanje pravougonika!"/>
    </Translations>
</DevRes>
```

- The identifier is used in the C++ code. For example, `td::String str(tr("open"));` will create a string ‘str‘ with the content "Otvori". If the English language is selected, then it will be "Open".
- The content of the translation will be visible within the application.

5.6.2 Defining Images via XML

Since images are not dependent on translation, their configuration file is located within the `res` directory.

- The root node must have the name `DevRes`, just like with translations.
- The root node can have two nested nodes:
  - `<Images>` for bitmap images (png) and
  - `<Symbols>` for vector images (symbols).
- To display images within the `natID` framework, the `gui::Image` class is used, and for symbols, `gui::Symbol`.
  - In both cases, if you want to use a resource, it is necessary to add a colon before the ID in the constructor, which tells the framework not to load from the system path but from the resources.

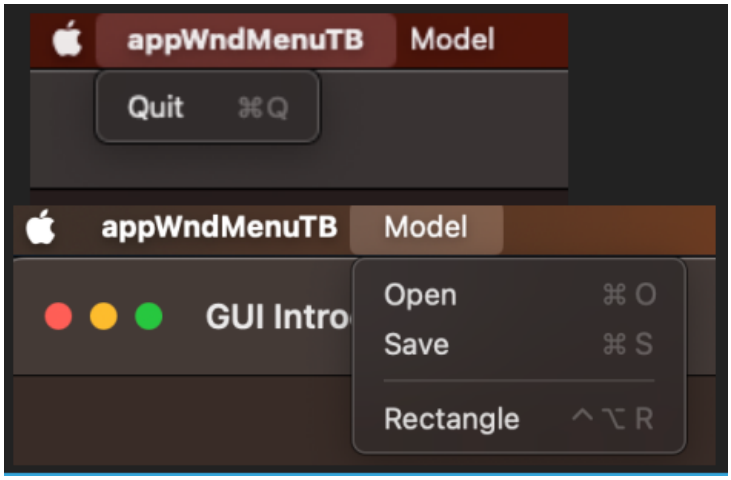
```
<?xml version='1.0'?>
<DevRes>
    <Images>
        <Res id="imgOpen" path="Work/Common/Icons/openSave/open64.png"/>
        <Res id="imgSave" path="Work/Common/Icons/openSave/save64.png"/>
    </Images>
    <Symbols>
        <Res id="roundRct" path="Work/Common/Symbols/Shapes/roundedrectangle.xml"/>
    </Symbols>
</DevRes>
```

- All paths within the configuration file are given relative to the HOME directory.

5.7 Creating Menus

- For the menu’s needs, a new `MenuBar` class is created, whose instance is located within `MainWindow`.

- A similar thing is done for the toolbar (the `ToolBar` class - will be explained a little later).
- Let's assume we need to create a menu that consists of two parts (submenus):
  - The first part will have the option to close the application (the "Quit" option).
  - The second part will have three options: "Open", "Save", "Rectangle".
    * The first two options in this submenu (Open and Save) can be triggered with the Ctrl+O and Ctrl+S keyboard shortcuts, respectively.
    * The last option (Rectangle) can be triggered with the Ctrl+Alt+R shortcut.



Slika 102: Menu appearance on macOS. On the Windows system, the menu is part of the window.

5.7.1 Implementing the Menu

- The `MenuBar` class implements the desired menu and is inherited from `gui::MenuBar`.
  - It has two `gui::SubMenu` instances, the first of which has one and the second has three menu options (choices).
- The `MenuBar` constructor requires entering the number of (sub)menus to be generated (line 30).
- The first (sub)menu is instantiated on line 31. The first parameter in the constructor is the menu ID (10), the second parameter is the menu name "testApp", and the third parameter is the number of options within that menu (one).
- The second (sub)menu is instantiated on line 32. The first parameter in the constructor is the menu ID (20), the second parameter is the menu name "Model", and the third parameter is the number of options within that menu (three).
- Line 34 calls the method that populates the first (sub)menu.
- Line 35 calls the method that populates the second (sub)menu.
- Lines 36 and 37 store the menu pointers in the base class infrastructure.
- Line 38 calls the base class method that completes the menu.

```
#pragma once
#include <gui/MenuBar.h>
class MenuBar : public gui::MenuBar
{
private:
    gui::SubMenu subApp;
    gui::SubMenu subFile;
public:
    MenuBar()
    : gui::MenuBar(2) // two menus
    , subApp(10, "testApp", 1)
    , subFile(20, "Model", 3)
    {
        populateSubAppMenu();
        populateSubFileMenu();
        _menus[0] = &subApp;
        _menus[1] = &subFile;
        prepare();
    }
//...
};
```

5.7.2 Populating the Menus

Populating the first (sub)menu is realized as follows:

```
void populateSubAppMenu()
{
    auto& items = subApp.getItems();
    items[0].initAsQuitAppActionItem(tr("Quit"), "q");
}
```

- Only one option for closing the application has been entered (can be activated with Ctrl+Q).

Populating the second (sub)menu is realized as follows:

```
void populateSubFileMenu()
{
    auto& items = subFile.getItems();
    items[0].initAsActionItem(tr("open"), 10, "o"); //Ctrl+O
    items[1].initAsActionItem(tr("save"), 20, "s"); //Ctrl+S
    items[2].initAsSeparator();
    items[3].initAsActionItem(tr("rect"), 30, "<Ctrl><Alt>r"); //Ctrl+Alt+R
}
```

- This (sub)menu has three "normal" items (options) that are created by calling the `initAsActionItem` method.
- The first parameter is the text (translation), the second is an identifier that we will need to recognize that the option was clicked, and the third parameter is optional and represents the shortcut.

5.8 Creating the Toolbar

Like the menu, the toolbar is part of `MainWindow`. The complete implementation of the toolbar is provided in the `ToolBar.h` header file.

```
#pragma once
#include <gui/ToolBar.h>
#include <gui/Image.h>
#include <gui/Symbol.h>

class ToolBar : public gui::ToolBar
{
protected:
    gui::Image _imgOpen;
    gui::Image _imgSave;
    gui::Symbol _symbol;
public:
    ToolBar()
    : gui::ToolBar("mainTB", 4) // ID and number of items
    , _imgOpen(":imgOpen")
    , _imgSave(":imgSave")
    , _symbol(":roundRct")
    {
        addItem(tr("open"), &_imgOpen, tr("openTT"), 20, 0, 0, 10);
        addItem(tr("save"), &_imgSave, tr("saveTT"), 20, 0, 0, 20);
        addSpaceItem();
        addItem(tr("rect"), &_symbol, tr("rectTT"), 20, 0, 0, 30);
    }
};
```

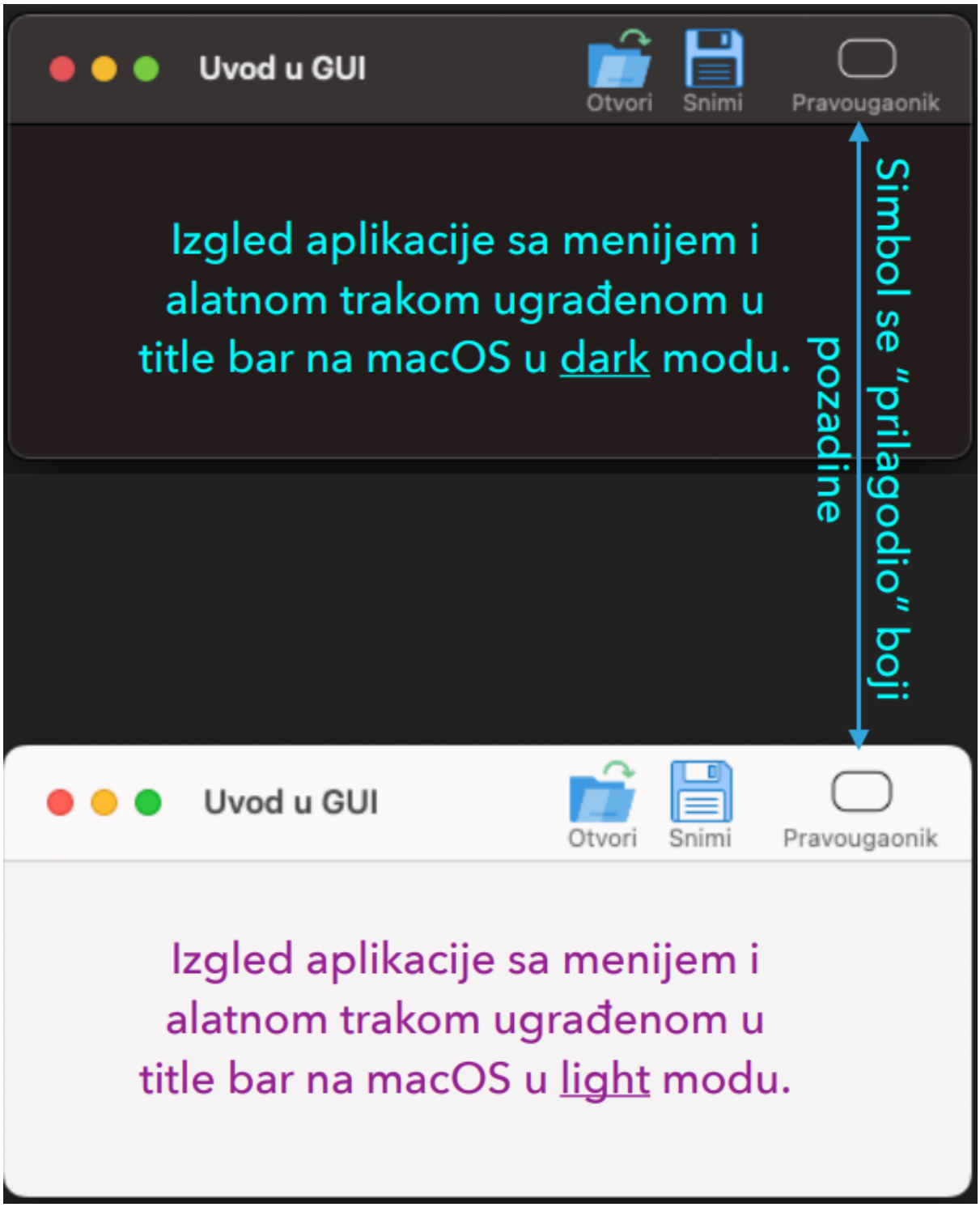
- The toolbar constructor has two parameters: 1. an identifier needed by the system and 2. the number of toolbar elements (4 in this case).
- Images and symbols are loaded from resources (':' at the beginning).
- The `addItem` method initializes one toolbar element with the following parameters: 1. item position, 2. identifier needed by the system, 3. text to be displayed on the track below the button, 4. pointer to the image or symbol, 5. tooltip, 6-8. identifier needed to recognize the item.
- A space is generated between toolbar items.

5.9 Reacting to Events

- The code so far has created an application that can work in two languages and has a menu and a toolbar.
- Modern graphical applications are based on the principle of handling "events" that are generated after clicking on a menu/toolbar option or another element.
- Reacting to events in the `natID` framework is very simple.
- All classes that directly or indirectly inherit the `Consumer` class have the ability to react to all events that the application user generates with their activities.
- The `gui::Window` class inherits the `gui::Frame` class, which in turn inherits the `gui::Consumer` class.

  - The `gui::Consumer` class defines a set of virtual methods for "consuming" events that descendant classes can overload.

- To react to menu and toolbar events, it is necessary to implement the virtual method `onActionItem` inside the `MainWindow` class.



Slika 103: Application appearance with menu and toolbar embedded in the title bar on macOS in dark and light modes. The symbol has adapted to the background color.

5.9.1 Consumer Methods

The consumer methods that allow event consumption are given in the header file `<gui/Consumer.h>`. It is necessary to overload only the method (or methods) depending on which event(s) you want to consume.

```
//for menus and toolbars
virtual bool onActionItem(gui::ActionItemDescriptor& aiDesc);
virtual bool onContextMenuUpdate(td::BYTE menuID, gui::ContextMenu* pMenu);
//for button clicks
virtual bool onClick(gui::Button* pBtn);
virtual bool onClick(gui::DrawableButton* pDrawableBtn);
//... and many more for different UI elements
virtual bool onAnswer(td::UINT4 questionID, gui::Alert::Answer answer);
virtual bool onBeginEdit(gui::LineEdit* pCtrl);
virtual bool onActivate(gui::LineEdit* pCtrl); //pressed enter
virtual bool onChangedSelection(gui::ComboBox* pCmb);
virtual bool onChangedValue(gui::Slider* pSlider);
virtual bool onTimer(gui::Timer* pTimer);
virtual bool onServerSocket(const gui::no::Notification& notification);
```

- **Note:** All operations that include interaction with graphical interface elements must be executed in the main program thread (main thread).
- Communication with threads is done through special `natID` asynchronous commands that call methods and execute them in the main thread. In this way, the workload can be executed in parallel. More on this later.

5.9.2 Reacting to Menu and Toolbar Actions

To correctly react to events generated by the menu and toolbar, it is necessary to use the identifiers used when creating the menu and toolbar elements.

- In the case of the toolbar, these are the last 4 parameters.
- In the case of the menu, the ID of the first menu, the first submenu, the last submenu, and the ID of the item itself are taken.
  - In our case, the first and last submenus had id=0, so the code for reacting to the activation of the first option of the second menu (and the first option of the toolbar) is given in the image below.
- If you react to an event, you need to return `true`. Otherwise, the `natID` framework continues to search for potential consumers of the event.

```
bool onActionItem(gui::ActionItemDescriptor& aiDesc) override {
    auto [menuID, firstSubMenuID, lastSubMenuID, actionID] = aiDesc.getIDs();
    auto pAI = aiDesc.getActionItem();
    if (menuID == 20 && firstSubMenuID== 0 && lastSubMenuID == 0)
    {
        switch (actionID)
        {
            case 10: // Open
            {
                td::String msgText(tr("mOpen"));
```

```
                td::String informativeText;
                informativeText.format("Handled onActionItem(subMenuID=%d, "
                    "firstSubSubMenuID=%d, lastSubSubMenuID=%d, "
                    "actionID=%d)", menuID, firstSubMenuID,
                    lastSubMenuID, actionID);
                showAlert(msgText, informativeText);
                return true;
            }
        }
    }
    // Handle other actions...
    return false; // Did not handle this event
}
```

6 Data Entry and Processing via Graphical User Interface

6.1 Arrangement of Elements for Data Editing (View, Layout)

- The previous presentation explained the creation of the main window, menus, and toolbars.
- For data editing, it is necessary to use so-called "Views" on the data.
  - To edit data in the window, it is necessary to instantiate an object of the `gui::View` class, which will be set as the data view within the window.
  - In the case of graphical drawing, the `gui::Canvas` class is used instead of the `View` class (more on this later).
- Data views are placed in the central part of the window.
- The window can have a variable size, so it is recommended that the content of the data view also reacts to changes in the window's size.
  - Since `natGUI` requires every View to react to window size changes, several classes have been created for this purpose that inherit/extend the `gui::Layout` class.

6.1.1 Common Layouts

The three most commonly used layouts are:

- `gui::HorizontalLayout` - arranges graphical elements one after another horizontally. If the width of the window/view increases, the extra width will be taken by the elements within this layout that have the ability to change width (almost all except `gui::Label`).
- `gui::VerticalLayout` - arranges graphical elements one after another vertically. If the height of the window/view increases, the extra height will be taken by the elements within this layout that have the ability to change height (this ability is not available for many elements).
- `gui::GridLayout` - arranges graphical elements in a matrix form. If the width or height of the window increases, the extra width/height will be taken by the elements within this layout that have the ability to change width/height.
  - The elements in this layout are aligned by rows and columns.

6.1.2 Arrangement of Elements through Composition

- A layout only serves to correctly arrange elements in the view.
- To display/edit data, it is necessary to instantiate objects of the appropriate classes for displaying and editing data.
  - All elements that have the ability to display/edit data are inherited from the base class `gui::Control`.
- It is important to note that both the `gui::View` class and the `gui::Layout` class are inherited from the `gui::Control` class.
- This achieves compositionality in GUI application design:

  - A View must have a Layout.
  - A Layout can have one or more elements, which can be:
    * Simple controls for displaying and editing data.
    * Another layout.
    * Another view.

6.2 Data Entry Elements

6.2.1 Label and LineEdit

- The `gui::Label` class is used to display a line of text without the possibility of editing.
  - This is the most commonly used class, whose width and height are adjusted to the given text.
  - It does not react to changes in the window's dimensions.
- The `gui::LineEdit` class is used for editing text in a single line.
  - It reacts to changes in the window's width.
- The `gui::LineEdit` class allows interaction with other parts of the application in three ways:
  - Manually retrieving the entered content using the `getText` method, which is located in the parent class `gui::TextCtrl`.
  - Manually setting the content using the `setText` method.
  - Reacting to events generated by the `gui::LineEdit` class by re-implementing the virtual methods `onBeginEdit` and `onFinishEdit` (defined in `gui::Consumer`) if this is enabled (achieved by calling `enableMessaging`, which is not enabled by default for this control).
    * `onBeginEdit` will be called after the first change in the control's content.
    * `onFinishEdit` will be called after the control loses focus.
- If event generation for the start and end of editing is enabled, the control redirects the call to the first parent instance of the `View` class.

6.2.2 NumericEdit

Editing of numerical data is done in instances of the `gui::NumericEdit` class. This class allows editing of most numerical data, which is defined by its constructor:

```
1 NumericEdit(td::DataType dataType,
2     LineEdit::Messages sendMsg = LineEdit::Messages::DoNotSend,
3     bool bShowThSep = true,
4     const td::String& toolTip = "",
5     int nDec = -1);
```

- `dataType` - defines the type of data being edited.
- `sendMsg` - defines whether `onBeginEdit` and `onFinishEdit` messages will be generated.
- `bShowThSep` - defines whether a thousands separator will be displayed.

- `toolTip` - defines the tooltip.
- `nDec` - defines to how many decimal places the data should be edited and is used only if the data is of C++ type `float` or `double` (`td::real4`, `td::real8`).

The `gui::NumericEdit` class allows interaction with other parts of the application in three ways (similar to `LineEdit`, which it inherits):

- Manually retrieving the entered content using one of the two `getValue` methods.
- Manually setting the content using the `setValue` method.
- Reacting to events generated by the `gui::LineEdit` class by re-implementing the virtual methods `onBeginEdit` and `onFinishEdit` (defined in `gui::Consumer`) if this is enabled (achieved by calling `enableMessaging`, which is not enabled by default for this control).
  - `onBeginEdit` will be called after the first change in the control's content.
  - `onFinishEdit` will be called after the control loses focus.
- If event generation for the start and end of editing is enabled, the control redirects the call to the first parent instance of the `View` class (identical to `LineEdit`).

Since the `gui::NumericEdit` class allows editing of different numerical data types, the `setValue` and `getValue` methods use `td::Variant` for exchanging numerical values with the class.

- The `getValue` method is implemented in the `gui::NumericEdit` class:

```
virtual bool getValue(td::Variant& val, bool checkType = false) const;
td::Variant getValue() const;

```

- The `setValue` method is implemented in the `gui::DataCtrl` class:

```
virtual bool setValue(const td::Variant& val, bool sendMessage=true);

```

- Getting (and setting) values in other elements that have the ability to edit/display data values is also done through the `td::Variant` class.
- In addition, `td::Variant` is used extensively in interactions with databases.
- Therefore, we will explain the functionality that `td::Variant` provides.

6.3 Polymorphic Type - `td::Variant`

The `td::Variant` class allows storing different data types:

- Logical: `td::boolean`
- Integer numeric: `td::byte, td::word, td::int2, td::uint2, td::int4, td::uint4, td::lint8`
- Numeric-financial (fixed number of decimal places): `td::decimal0, td::decimal1,..., td::decimal4`

- Numeric with floating-point (float and double): `td::real4, td::real8`
- Date/time: `td::date, td::time, td::dateTime`
- Textual: `td::string8`
- Other types:
  - Colors: `td::colorID, td::color`
  - Line type: `td::linePattern`
  - Pointers
  - ...

In interaction with objects of the `gui::NumericEdit` class, the user selects the data type to be edited in the constructor.

- With the same data type that is set in `td::Variant`, the user can set the value within `gui::NumericEdit` using the `setValue` method.
- If the `td::Variant getValue() const;` method is used, the user gets the value and data type within the returned `td::Variant` instance.
- This approach enables an identical approach to data exchange with most controls.
- It only remains to clarify the way of setting the desired type and value of the data through an object of the `td::Variant` class.

Setting the data type to be stored within a `td::Variant` object can be done during its creation:

```
td::Variant varValue1(td::int4);
```

After that, its value can be set from data of the appropriate type:

```
td::INT4 val1 = 5;
varValue1.setValue(val1);
```

- The data type of the parameter `val1` must be identical to the data type currently held by `varValue1`, otherwise the value of `varValue1` will remain unchanged.

The same effect would be achieved if the `td::Variant` object was created in the following way:

```
td::INT4 val2 = 5;
td::Variant varValue2(val2); // in this case, both type and value are taken
```

If a value is set to a `td::Variant` object using the assignment operator (=), then it takes on both the new value and the new data type, regardless of the value and data type it had before the assignment operation.

```
td::String str("Some string");
varValue1 = str; // after this operation, varValue1 has type td::string8
```

- In case of interaction with databases, the constructor for initializing string types allows the input of two additional parameters that are used for interaction with databases (the type used in the database and the length of the string in the database). See the Cpp and DB example.

Information about the data type stored within a `td::Variant` object can be obtained using the `getType` method. E.g.:

```
td::DataType dt = varValue2.getType();
```

Retrieving the stored value from within a `td::Variant` object can be done in several ways. The two most commonly used are:

- Using the `getValue` method, which requires a parameter of a type identical to the one stored in the `td::Variant` object.

```
td::INT4 val;
varValue1.getValue(val);

```

  - If `varValue1` is of type `td::int4`, then the call to `getValue` will assign the value stored in the variant to the variable `val`.
  - In some cases, it is possible to retrieve the value of another type.

- By explicitly requesting the value of the desired type.
  - In this case, there is one get method for each supported type. The first mentioned variant is recommended.

Now that the way of working with the `td::Variant` class has been explained, we can continue with the explanation of the other elements.

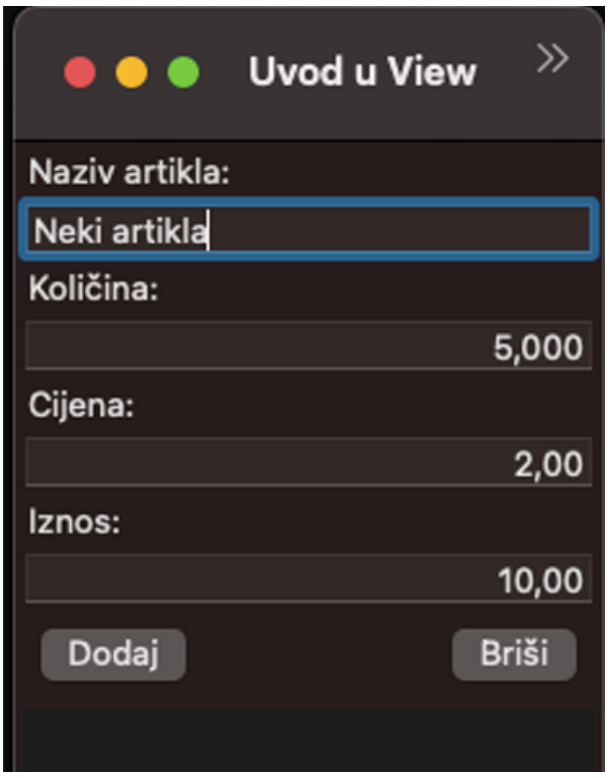
6.4 TextEdit and Button

- For the needs of the first example that creates a `gui::View` with certain elements, it is necessary to explain the elements `gui::TextEdit` and `gui::Button`.
- The `gui::TextEdit` class is used for editing multiple lines of text.
  - It changes both width and height within the set layout.
  - The content of the `gui::TextEdit` object can be set using the `setText` method.
  - The `appendString` method adds a string to the end of the content of the `gui::TextEdit` object.
- The `gui::Button` object allows the user to click with the mouse (or in another way) and thus call a certain activity within the application.
  - The `natID` framework in this case redirects the click event to the first parent instance of the `gui::View` class.
  - The processing of the button click event is then simply performed by implementing the virtual `onClick` method with the appropriate parameter:

```
virtual bool onClick(gui::Button* pBtn);

```

6.5 Example #1 - Vertical Layout

- To demonstrate the data processing elements so far, an application will be constructed that enters the item name, quantity (three decimals), and price (two decimals), and a field where the amount (two decimals) will be displayed. In addition to this, it is necessary to add a `TextEdit` that will display `onFinishEdit` events as well as the ability to write the current values into the `TextEdit` via the "Add" button. A button with the text "Delete" will be used to clear the content of the `TextEdit`.
- In the first version, the example will be done using a vertical layout. The "Add" and "Delete" buttons should be placed horizontally next to each other.
- The application should look like the image on the right.
- The "02_AppWndMenuTB" example from the previous lectures was used as a basis for the implementation.
  - The Menu and toolbar will not be used in the example and are left available for future use (experimentation).
- The implementation of this example is given in the project "01_DemoVert", which is in the attachment.
- **Note:** For the application to run in the development environment, it is necessary to enter the appropriate setting in `DevEnv/ResPaths.txt`.



Slika 104: Example application using a vertical layout. *For practice: Make the application so that the labels are on the left side of the edit controls.*

6.5.1 Implementation (MainWindow)

After adjusting the application name in the starting project (the new name is `appDemoVert`), it is necessary to correct the `MainWindow.h` file, which now needs to contain an instance of the `gui::View`

object. In this case, that instance is created as a class `ViewVert` that is inherited from `gui::View`.

```
#include <gui/Window.h>
#include "MenuBar.h"
#include "ToolBar.h"
#include "ViewVert.h" // NEW

class MainWindow : public gui::Window
{
protected:
    MenuBar _mainMenuBar;
    ToolBar _toolBar;
    ViewVert _mainView; // NEW
public:
    MainWindow()
    : gui::Window(gui::Size(200, 380))
    {
        setTitle(tr("appTitle"));
        _mainMenuBar.setAsMain(this);
        setToolBar(_toolBar);
        setCentralView(&_mainView); // NEW
    }
};
```

- The implementation of our "view" (`gui::View`) on the data is given in the header file `ViewVert.h`.
- As mentioned earlier, the window (in this case `MainWindow`) should have an instance of the data view, which is what was done with this line of code.
- This command (`setCentralView`) tells the window to use `_mainView` as the central data view. This places the instance of the `ViewVert` class (i.e., `_mainView`) in the central part of the window. The size of this view will adapt to the dimensions of the window (at the beginning and after every size change).

6.5.2 Implementation (ViewVert Header)

```
#pragma once
#include <gui/View.h>
#include <gui/Label.h>
#include <gui/LineEdit.h>
#include <gui/NumericEdit.h>
#include <gui/TextEdit.h>
#include <gui/Button.h>
#include <gui/HorizontalLayout.h>
#include <gui/VerticalLayout.h>
#include <cnt/StringBuilder.h>

class ViewVert : public gui::View
{
private:
protected:
    gui::Label _lblName;
    gui::LineEdit _editName;
    gui::Label _lblQuantity;
    gui::NumericEdit _neQuantity;
    gui::Label _lblPrice;
    gui::NumericEdit _nePrice;
```

```
22      gui::Label _lblValue;
23      gui::NumericEdit _neValue;
24      gui::Button _btnAdd;
25      gui::Button _btnDelete;
26      gui::TextEdit _textEdit;
27      gui::HorizontalLayout _hlButtons;
28      gui::VerticalLayout _vl;
29  };
```

- The `ViewVert` class is inherited from `gui::View`.
- `natID` header files are needed for the view implementation.
- The newly created `ViewVert` class inherits the `gui::View` class from the `natID` framework.
- Objects needed to create the visual elements.
- A horizontal layout is used to manipulate the buttons (Add, Delete).
- The vertical layout contains all the elements for editing, including the horizontal layout (shown on the next slide).

6.5.3 Implementation (ViewVert Constructor)

The constructor of the `ViewVert` class initializes the GUI elements.

```
1   ViewVert()
2   : _lblName(tr("name"))
3   , _lblQuantity(tr("quant"))
4   , _neQuantity(td::decimal3, gui::LineEdit::Messages::Send)
5   , _lblPrice(tr("price"))
6   , _nePrice(td::decimal2, gui::LineEdit::Messages::Send)
7   , _lblValue(tr("value"))
8   , _neValue(td::decimal2)
9   , _btnAdd(tr("add"))
10  , _btnDelete(tr("delete"))
11  , _hlButtons(3)
12  , _vl(10)
13  {
14      //set as read only
15      _neValue.setAsReadOnly();
16
17      //populate horizontal layout with buttons
18      _hlButtons.append(_btnAdd);
19      _hlButtons.appendSpacer();
20      _hlButtons.append(_btnDelete);
21
22      //populate vertical layout
23      _vl << _lblName << _editName << _lblQuantity << _neQuantity
24          << _lblPrice << _nePrice << _lblValue << _neValue
25          << _hlButtons << _textEdit;
26
27      setLayout(&_vl); //set the layout of this view
28  }
```

- `NumericEdit` in the constructor requires the data type (first parameter, mandatory) and optionally whether to send notifications for `onBeginEdit` and `onFinishEdit` (second parameter, optional).
- The horizontal layout requires the number of elements it controls in the constructor (in this case 3: Add button, spacer, and Delete button).
- The vertical layout requires the number of elements it controls in the constructor (in this case 10).
- Populating the horizontal layout (3 elements).
- Populating the vertical layout (10 elements).
- Setting the vertical layout for the `ViewVert` view.

6.5.4 Implementation (ViewVert Event Handling)

Changes in the quantity and price values require calculation and display of the calculated value for the item (which is set in "read only" mode and cannot be modified by the user). As stated earlier, it is necessary to implement the `onFinishEdit` method as follows:

```
bool onFinishEdit(gui::LineEdit* pCtrl) override
{
    if ( (pCtrl == &_neQuantity) || (pCtrl == &_nePrice) )
    {
        // Retrieve edited values for quantity and price.
        td::Variant quant = _neQuantity.getValue();
        td::Variant price = _nePrice.getValue();

        // The multiplication operator of two numeric values of different
        // types returns a double type.
        auto value = quant * price;

        // The r8Val method takes the double value stored in the 'value' var.
        td::Decimal2 decVal(value.r8Val());

        // Set the calculated value (amount = quantity * price)
        // in the editing element which is of type td::decimal2.
        _neValue.setValue(decVal);
        return true;
    }
    return false;
}
```

It is also necessary to handle the user requests that are received after clicking the Add and Delete buttons. For this purpose, it is necessary to implement the `onClick` method as follows (override the virtual `onClick` method defined in the `Consumer` class):

```
bool onClick(gui::Button* pBtn) override
{
    if (pBtn == &_btnAdd)
    {
        // Handling the request received after clicking the Add button.
        // A string is formed consisting of the item name, quantity, and value.
        cnt::StringBuilderSmall sb;
        sb.appendString(_editName.getText());
        sb.appendCString(", Quantity=");
        sb.appendString(_neQuantity.getText());
```

```
        sb.appendCString(", Value=");
        sb.appendString(_neValue.getText());
        sb.appendCString(" ");
        td::String str = sb.toString();

        // The resulting string is then added to the TextEdit field.
        _textEdit.appendString(str);
        return true;
    }
    else if (pBtn == &_btnDelete)
    {
        // Handling the request received after clicking the Delete button.
        // The clean method is called which clears the content of the TextEdit.
        _textEdit.clean();
        return true;
    }
    return false;
}
```

In addition to using virtual methods that route events for a group of objects through a single method (`onClick` in this case), it is also possible to process events using lambda functions. For objects that have a defined event handler via a lambda function, the event will not be routed to the common virtual method.

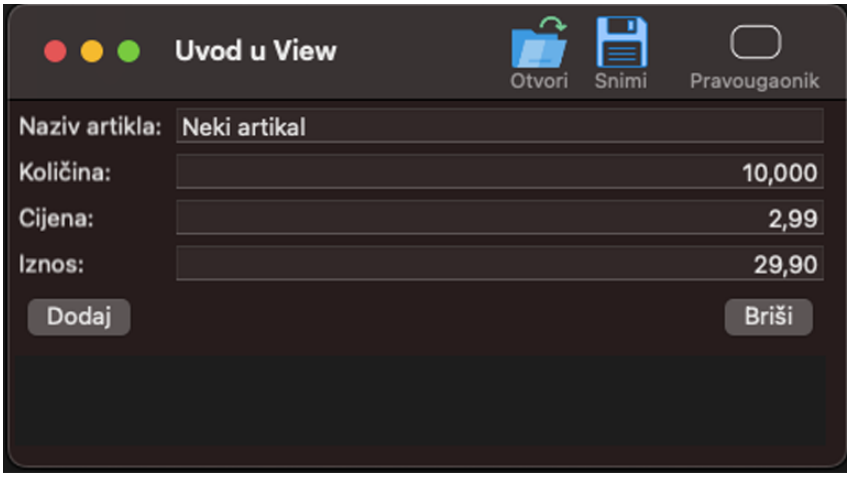
If, for example, the following lines of code are added to the `ViewVert` class constructor, the event processing would be done per object via lambda methods, and the corresponding virtual methods for these objects would not be called:

```
// use lambdas for handling events per object
_btnAdd.onClick([this]()
{
    cnt::StringBuilderSmall sb;
    sb.appendString(_editName.getText());
    sb.appendCString(", Quantity=");
    sb.appendString(_neQuantity.getText());
    // ... and so on
    _textEdit.appendString(str);
});

_btnDelete.onClick([this]()
{
    _textEdit.clean();
});
```

6.6 Example #2 - Grid Layout

This example demonstrates the same functionality as the previous one, but arranges the input elements in a matrix form using `GridLayout`. The rest of the logic remains the same.

- The implementation is based on the "01_DemoVert" project.
- The implementation of this example is provided in the "02_DemoGrid" project attached.



Slika 105: The same application implemented with a GridLayout.

6.6.1 Implementation (MainWindow)

The only change in `MainWindow.h` is replacing the `ViewVert` instance with a `ViewGrid` instance. The rest of the code is unchanged.

```
#include <gui/Window.h>
#include "MenuBar.h"
#include "ToolBar.h"
#include "ViewGrid.h" // NEW

class MainWindow : public gui::Window
{
protected:
    // ...
    ViewGrid _mainView; // NEW, instead of ViewVert
public:
    MainWindow()
    : gui::Window(gui::Size(400, 280))
    {
        // ...
        setCentralView(&_mainView);
    }
};
```

6.6.2 Implementation (ViewGrid Header)

The header for `ViewGrid` is very similar to `ViewVert`, with the key difference being the use of `GridLayout` instead of `VerticalLayout`.

```
#pragma once
#include <gui/View.h>
// ... other includes
#include <gui/GridLayout.h> // NEW

class ViewGrid : public gui::View
{
// ... member variables are the same ...
protected:
    // ...
```

```
    gui::HorizontalLayout _hlButtons;
    gui::GridLayout _gl; // NEW, instead of VerticalLayout
};
```

6.6.3 Implementation (ViewGrid Constructor)

The constructor initializes the GUI elements and populates the `GridLayout` directly.

```
ViewGrid()
: _lblName(tr("name"))
// ... other initializations ...
, _hlButtons(3)
, _gl(6, 2) // Grid with 6 rows and 2 columns
{
    // ... setAsReadOnly and populate _hlButtons ...

    // populate grid layout
    _gl.insert(0, 0, _lblName);     _gl.insert(0, 1, _editName);
    _gl.insert(1, 0, _lblQuantity); _gl.insert(1, 1, _neQuantity);
    _gl.insert(2, 0, _lblPrice);    _gl.insert(2, 1, _nePrice);
    _gl.insert(3, 0, _lblValue);    _gl.insert(3, 1, _neValue);
    _gl.insert(4, 0, _hlButtons, -1); // span all columns (-1)
    _gl.insert(5, 0, _textEdit, -1);  // span all columns (-1)

    setLayout(&_gl); // set the layout of this view
}
```

- The third parameter of `insert`: -1 means span from the inserted column to the end; a positive number means span that many consecutive columns.

6.6.4 Alternative Implementation (GridComposer)

An alternative approach to populating a `GridLayout` uses the `gui::GridComposer` class.

```
// inside ViewGrid constructor
{
    // ... initializations and _hlButtons population ...

    // using GridComposer (appendRow & appendCol)
    gui::GridComposer gc(_gl);
    gc.appendRow(_lblName);     gc.appendCol(_editName);
    gc.appendRow(_lblQuantity); gc.appendCol(_neQuantity);
    gc.appendRow(_lblPrice);    gc.appendCol(_nePrice);
    gc.appendRow(_lblValue);    gc.appendCol(_neValue);
    gc.appendRow(_hlButtons, 0); // span columns until the last (0)
    gc.appendRow(_textEdit, 0);  // span columns until the last (0)

    setLayout(&_gl); //set the layout of this view
}
```

- `appendRow` increments the current row index and sets the column index to 0.
- `appendCol` increments the column index for the current row.

- The second parameter specifies how many columns the element spans. A value of 0 means span across all remaining columns.

6.7 Other Data Processing Elements

Other data processing elements work on a similar principle to the elements covered in this lecture.

- All events that can be processed are implemented as virtual methods defined in the `gui::Consumer` class.
- The implementation of events is done not on the editing class itself, but on the view, because all other objects are available on the view, which facilitates data processing.
- If necessary, events can be redirected to any other consumer using the `forwardMessagesTo` method.
- Most elements support getting and setting values using `td::Variant`:

```
virtual bool setValue(const td::Variant& val, bool sendMessage=true);
virtual bool getValue(td::Variant& val, bool checkType = false) const;

```

- For simpler work, in addition to these two methods, most controls have special (dedicated) methods for getting/setting values.

6.8 `gui::ComboBox`

The `gui::ComboBox` allows the selection of one element from a list of allowed items.

Appearance:

- The ComboBox can be populated using the following methods:

```
void addItem(const td::String& item);
void addItem(const char* item);
void addItems(const td::String* pItems, size_t nItems);

```

- The currently selected element can be obtained by calling the `getSelectedIndex` method, which returns -1 if the combo box is empty, otherwise the 0-based index.
- The position of the currently selected element can be changed programmatically or through user interaction. Programmatic setting is done by calling `selectIndex`.
- If the user changes the currently selected element, `natID` calls the virtual method `onChangedSelection`:

```
virtual bool onChangedSelection(gui::ComboBox* pCmb);

```

6.9 gui::Slider

The gui::Slider allows the selection of a numerical value within given limits by visually moving a slider.

Appearance:

- The value that can be edited is of type double.
- The range can be set using the setRange method. If not used, the default range is 0 to 1.
- The current value can be retrieved with getValue.
- The slider's position can be changed programmatically with setValue.
- If the user changes the slider's position, natID calls the virtual method onChangedValue:

```
1 virtual bool onChangedValue(gui::Slider* pSlider);
2
```

6.10 gui::ProgressIndicator

The gui::ProgressIndicator allows displaying a numerical value (progress) within given limits.

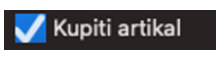
Appearance:

- The value displayed is of type double.
- The range must be between 0 and 1.
- The current value can be retrieved with getValue.
- The indicator's position can only be changed programmatically with setValue.
- Since the user cannot change the indicator's value via the GUI, natID does not generate any events.

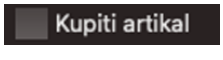
6.11 gui::CheckBox

The gui::CheckBox allows editing of a logical (bool) value.

Appearance (true):



Appearance (false):



- It can be constructed with included text (as above) or without text.
- The current value can be retrieved with `isChecked`.
- The value can be changed programmatically or through user interaction. Programmatic setting is done with `setChecked`.
- If the user clicks the CheckBox, `natID` calls the virtual method `onClick`:

```
virtual bool onClick(gui::CheckBox* pBtn);

```

6.12 gui::DateEdit and gui::TimeEdit

These controls allow editing of date and time values.

Appearance: 

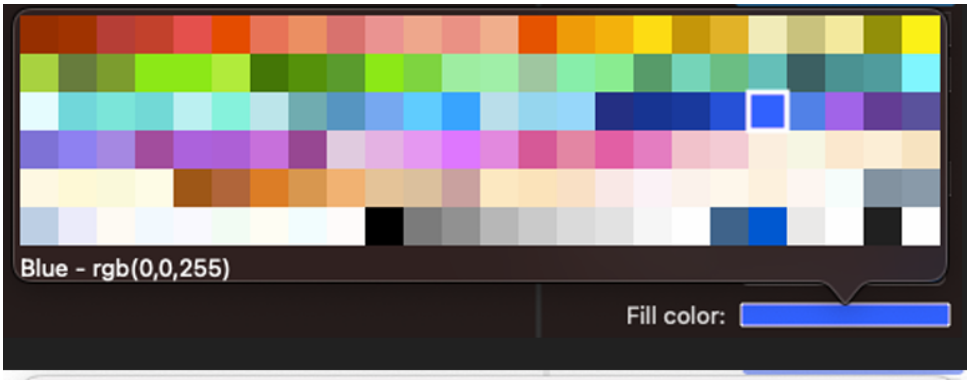
- The current value can be retrieved with `getValue`.
- The value can be changed programmatically or through user interaction. Programmatic setting is done with `setValue`.
- If the user makes a change (and the control loses focus), `natID` first checks for a defined lambda handler via `obj.onChangedValue(lambdaHandler)`.
- If no lambda handler is defined, `natID` calls the virtual method `onChangedValue`:

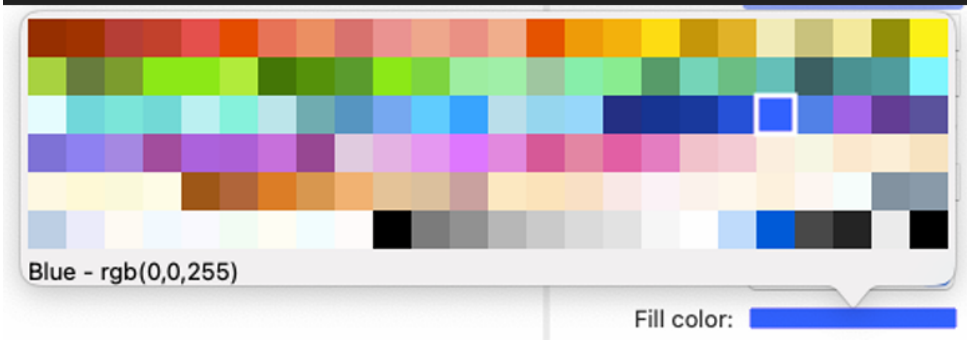
```
virtual bool onChangedValue(gui::DateEdit* pDE);
virtual bool onChangedValue(gui::TimeEdit* pTE);

```

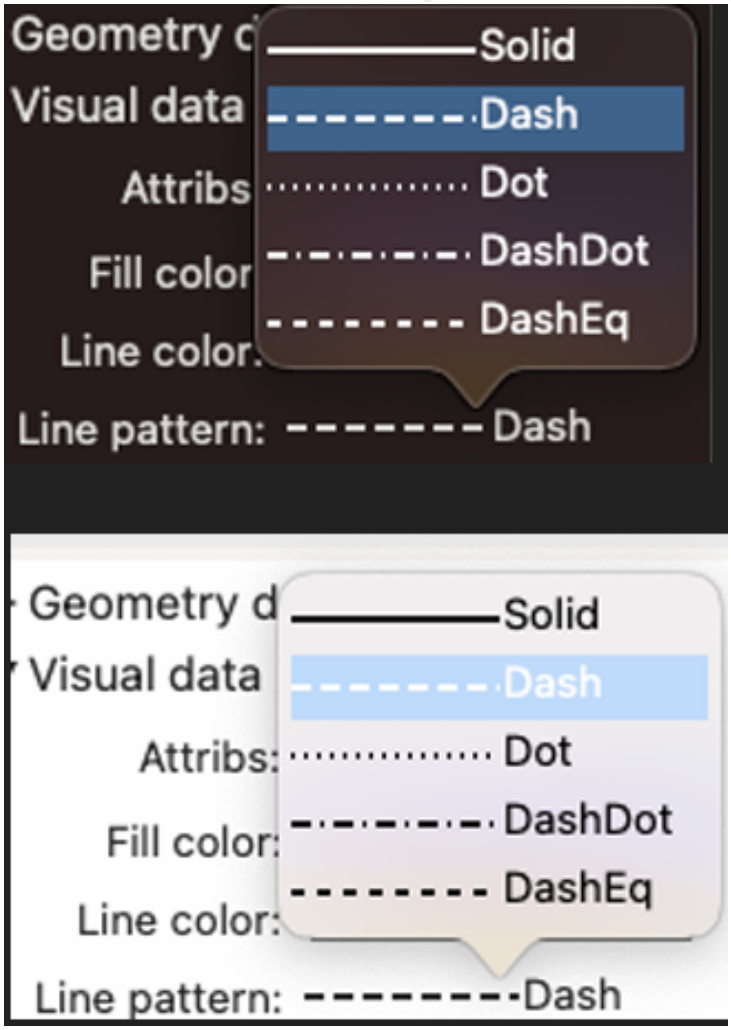
6.13 gui::ColorPicker and gui::LinePatternPicker

These complex elements allow the user to select a color or a line pattern from a predefined palette.

Appearance:







- The first click does not generate a user event but causes a popover view to be displayed showing the palette.

- The popover view is closed with the next click (anywhere). If the click is on a valid choice, an event is generated.
- The current value is retrieved with `getValue`.
- The value is set programmatically with `setValue`.
- If the user makes a change, the event is handled via a lambda or the virtual method `onChangedValue`:

```
virtual bool onChangedValue(gui::ColorPicker* pCP);
virtual bool onChangedValue(gui::LinePatternPicker* pLPP);

```

6.14 Other Controls

6.14.1 `gui::PasswordEdit`

Allows editing of a single line of text without displaying the content to the user.

Appearance:

- This class is inherited from `gui::LineEdit` and has an identical way of communicating with the code.

6.14.2 `gui::ImageView`

Allows displaying images (`gui::Image`) and symbols (`gui::Symbol`) on the user interface.

Appearance:

- The constructor of this class allows maintaining the aspect ratio when the view dimensions change (the `keepRatio` parameter).
- Programmatic setting of the image/symbol is done by calling the `setImage` method.

```
void setImage(const Image* img);
void setImage(const Symbol* pSymb);

```

7 Editing Tabular Data and Interaction with Databases

7.1 Tabular Data Editing

- Editing tabular data, or data that can be represented as a vector of structures, is most suitably done using the `gui::TableEdit` element.
- `gui::TableEdit` does not allow for so-called "in-place" editing.
  - Instead, it displays the data with appropriate formatting in a tabular form.
- In principle, `gui::TableEdit` serves only for display, i.e., as a view of the data.
- The data itself that is displayed is held in a `dp::IDataSet` interface.
  - `dp::Recordset` was partially covered in the "Introduction to Databases" chapter.
- To display the data, it is necessary to specify the desired columns and the data formatting method.
- All or part of the data that the `dp::IDataSet` interface makes available can be shown.
- Columns can be set in any order, independent of the order within the `dp::IDataSet`.
- The order and width of columns can be changed during the application's execution (run-time).
- It is important to emphasize that `dp::IDataSet` does not have to be created from a database. It can be implemented using any other data source.
- For the demonstration, a SQLite database will be used.

7.1.1 Project Setup

For the demonstration, the project "01_DemoTableEdit" will be used.

- The project is created based on the "02_DemoGrid" project from the previous lectures (copy-paste then adjustments).
- Since this project uses SQLite, it is necessary to correct the CMake file to include the `natDP` (DataProvider) library.
- For this reason, it is necessary to include the following line in `CMakeLists.txt`:

```
include(${WORK_ROOT}/DevEnv/DataProvider.cmake)

```

- Also, in the edited `.cmake` file, it is necessary to include the desired libraries for interaction with databases (`DP_LIB_DEBUG` and `DP_LIB_RELEASE`):

```
target_link_libraries(${APPDEMOTABLEEDIT_NAME}
    debug ${MU_LIB_DEBUG}
    debug ${NATGUI_LIB_DEBUG}
    debug ${DP_LIB_DEBUG}
    optimized ${MU_LIB_RELEASE}
    optimized ${NATGUI_LIB_RELEASE}
    optimized ${DP_LIB_RELEASE}
)

```

7.1.2 Database Schema

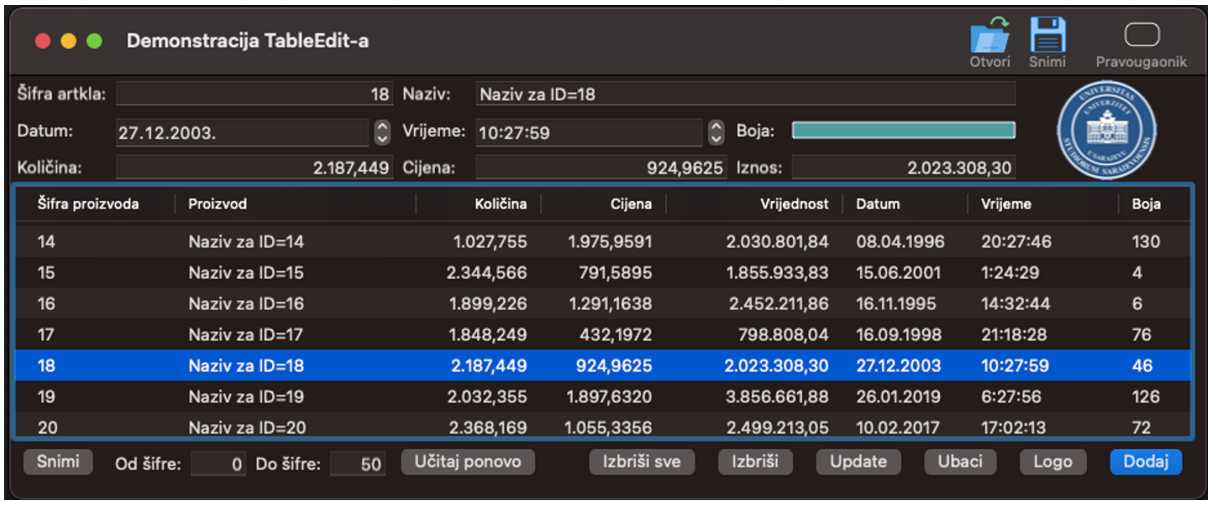
To better understand the editing of tabular data, a simple application will be created that edits tabular data from a table created in SQLite as follows:

```
CREATE TABLE "Test" (
    "ID"     INTEGER NOT NULL,
    "Name"   TEXT NOT NULL,
    "Quant"  INTEGER NOT NULL,
    "Price"  INTEGER NOT NULL,
    "Date"   INTEGER NOT NULL,
    "Time"   INTEGER NOT NULL,
    "R4"     REAL NOT NULL,
    "R8"     REAL NOT NULL,
    "Color"  INTEGER NOT NULL,
    PRIMARY KEY("ID")
);
```

- **ID** - represents the item's identification number (code).
- **Name** - represents the item's name.
- **Quantity** - item quantity (`td::decimal3`).
- **Price** - item price (`td::decimal4`).
- **Date** - item production date.
- **Time** - item production time.
- **Color** - item color.
- **R4** and **R8** will be used in one of the following demonstrations.

7.1.3 Application Interface

The table is edited in the application with the following appearance:



Slika 106: The demonstration application for editing tabular data.

Q: What layout is used to implement the complete view?

- **A:** A `GridLayout` with the `gui::TableEdit` element currently selected (bordered by a blue line).

Q: How many rows and columns does the `GridLayout` from the image above have?

- **A:** 5 rows and 7 columns. In the last row (the row with the buttons and the from/to ID fields), a `HorizontalLayout` is used.

7.1.4 Application Logic

- The application edits data stored in the `Test` table by displaying it in the central `TableEdit` field with column names as previously shown.
- If the user changes the selected row (by mouse click or another gesture), the column values from the selected row are transferred to the elements above the `TableEdit` element, where the values can be changed.
- After that, the edited values can be:
  - Added as a new row by clicking the "Add" button.
  - Inserted before the selected row by clicking the "Insert" button.
  - Written to the selected row by clicking the "Update" button.
- Deleting the selected row is achieved by clicking the "Delete" button, while deleting the entire content is done by clicking "Delete All".
- The edit fields "From ID" and "To ID", together with the "Reload" button, load data from the table.
- The "Save" button saves the data to the table (with the SQL Insert command).
- Pay attention that the Amount is not given in the table but is displayed in the corresponding `TableEdit` column.
  - The value of this field is not saved and does not consume memory but is calculated as needed using a delegate.

7.2 Implementation Steps

The complete method of editing tabular data is realized by implementing the following steps:

1. Create a corresponding View with a `TableEdit` element.
2. Assign a `dp::IDataSet` interface to the `TableEdit` element during the initialization process.
   - The task of the `dp::IDataSet` interface is to hold the data in memory during editing.
   - Data is loaded from the database into the `dp::IDataSet`, rows are changed, deleted, and new rows are added, all within the computer's memory, i.e., within the `dp::IDataSet` interface instance.
   - On request, the data from the `dp::IDataSet` can be saved to the database using the SQL INSERT command.

3. For each column visible in the `TableEdit` element, it is necessary to specify the exact column number within the `dp::IDataSet`.
   - We know this information because the `dp::IDataSet` is loaded with a SELECT SQL command with precisely specified column numbers that were bound during execution.
4. During operation, `gui::TableEdit` calls a virtual method after each change of the selected row:

```
virtual bool onChangedSelection(gui::TableEdit* pTE);

```

   - The application code should implement this method to transfer the values stored in the newly selected row to the editing elements.

7.3 Initializing `TableEdit` and Interacting with `IDataSet`

- `gui::TableEdit` requires a `dp::IDataSet` interface that provides it with data for display.
  - The `gui::TableEdit` itself is for visualization only and is not a data container.
- The complete implementation of the view is realized in the `ViewTableEdit` class (header file `ViewTableEdit.h`).
- A member variable has been created for communication with the database:

```
dp::IDatabasePtr _db;

```

- A member variable has been created as a container for the loaded data from the `Test` table:

```
dp::IDataSetPtr _pDS;

```

- The `dp::IDataSet` is loaded from SQLite in the manner explained earlier.
  - In this example, data loading is realized with the `populateData` method.

```
void populateData()
{
    fo::fs::path homePath;
    mu::getHomePath(homePath);
    fo::fs::path testDBPath = (homePath / "other_bin/TestData/natGUITest/Test.db");

    if (!_db->connect(testDBPath.string().c_str()))
        return;

    _pDS = _db->createDataSet("SELECT * from Test WHERE ID>=? and ID<=?",
                              dp::IDataSet::Execution::EX_MULT);
    dp::Params params(_pDS->allocParams());
    params << _paramFrom << _paramTo;

    _pDS->setDelegate(&_delegate);

    //specify columns to obtain from the data provider
    dp::DSColumns cols(_pDS->allocBindColumns(10));
    cols << "ID" << td::int4 << "Name" << td::string8
         << "Quant" << td::decimal3 << "Price" << td::decimal4
```

```
        << "Date" << td::date << "Time" << td::time
        << "R4" << td::real4 << "R8" << td::real8
        << "Color" << td::word
        << "Value" << dp::DSColumn::Type::CalcOnConsume << td::decimal2;

    if (!_pDS->execute())
    {
        //show log
        _pDS = nullptr;
        return;
    }
}
```

- In this case, the `IDataSet` can be loaded multiple times, so the parameter `dp::IDataSet::Execution::EX_MULT` is used. If the data is not loaded from the database, use `EX_NO`.
- Setting a "delegate" for calculating values that are not in the database. A delegate is essentially a pointer to a function.
- Each column must start with a name and end with a type. The first column has an index of 0.
- Specifying a column that is calculated in the "delegate". The delegate is called after every change of the selected row in `TableEdit`. In this case, the value is calculated during consumption (`CalcOnConsume`). If you want the calculation to be done only during loading, then `CalcOnLoad` is specified. In that case, memory space is reserved for each element as well as for "normal" columns.

Q: How do we know the exact index of the column that `TableEdit` requires?

7.3.1 `IDataSet` Delegate

A delegate needs to be used in the following cases:

- When creating queries requires complex operations (e.g., running sum).
  - Using a delegate, this is simply implemented and executed with the `CalcOnLoad` option.
- When we want to save space needed for storing data in the database, to reduce the amount of data needed for transfer (important if the server is accessed over a local network or the internet), and in cases where we want to reduce the required memory on the application side.
- When we want to reduce redundancy in the table.

The implementation of the delegate is very simple and requires the implementation of the `dp::IDataSetDelegate` interface, which has two methods (`onLoad` and `onConsume`):

```
namespace dp {
class IDataSetDelegate
{
public:
    virtual void init(dp::IDataSet* pDS)
    { }

    virtual bool onLoad(dp::IDataSet* pDS, dp::RowWrapper& row, size_t iRow)
    { return false; }
```

```
10
11      virtual bool onConsume(dp::IDataSet* pDS, dp::RowWrapper& row, size_t iRow)
12      { return false; }
13  };
14  } //namespace dp
```

- The `init` method will be called only once and is used for initialization. In most cases, the column indexes are taken from the `pDS` dataset using known column names. This achieves flexibility in the order of columns.
- The `onLoad` method is called once for each record (row) of the `pDS` dataset during loading. If it returns false, then it stops being called.
- The `onConsume` method is called for each newly selected record (row) of the `pDS` dataset. If it returns false, then this method stops being called.

In our example, only `CalcOnConsume` is used, so it is only necessary to implement the `onConsume` method:

```
1  class DSDelegate : public dp::IDataSetDelegate
2  {
3      int _indQuant = -1;
4      int _indPrice = -1;
5      int _indValue = -1;
6  public:
7      void init(dp::IDataSet* pDS) override
8      {
9          _indQuant = pDS->getColIndex("Quant");
10         _indPrice = pDS->getColIndex("Price");
11         _indValue = pDS->getColIndex("Value");
12         assert(_indQuant >= 0 && _indPrice >= 0 && _indValue >= 0);
13     }
14     bool onConsume(dp::IDataSet* pDS, dp::RowWrapper& row, size_t) override
15     {
16         td::Variant quant(row[_indQuant]);
17         td::Variant price(row[_indPrice]);
18         td::Variant value = quant * price;
19         td::Decimal2 decVal(value.r8Val());
20         row[_indValue].setValue(decVal);
21         return true;
22     }
23 };
```

- Elements that calculate on `onConsume` are calculated after every change of the selected row.
- `IDataSet` prepares the current row (`dp::RowWrapper& row`) which consists of `td::Variant` elements for each column.
- The `onConsume` method uses existing values in the current row and calculates the ones that are needed. In our case, the quantity and price elements (indexes 2 and 3) are used to calculate the value (index 9).
- In this way, the value is held only in the current row, which is temporarily held in memory.
- Elements that are calculated on `onLoad` will be calculated and saved in memory (in the `IDataSet`) during loading.

7.4 Finalizing `TableEdit` Initialization

After loading the data (or defining the columns in `IDataSet`), it is possible to map the visual `TableEdit` columns with the data columns within `IDataSet`.

- In our case, the `ViewTableEdit` class has the following member variable `gui::TableEdit _table;`. The initialization is realized in the method `void initTable(int visPopulateType);`.
- The binding (initialization) can be performed in three ways:
  - The first way takes all columns from the `IDataSet` and displays them in the `TableEdit`.
    - The column names used for the header of the `TableEdit` will be identical to those specified during loading. The initialization in this case is a call to the `init` method with the second parameter equal to zero:

```
1 _table.init(_pDS, 0);
2
```

  - The second way allows the selection of desired columns in the corresponding order.
    - In this case, a list of columns is passed to the `init` method as the second parameter:

```
1 _table.init(_pDS, {0, 1, 2, 3, 9, 4, 8});
2
```

- The first two approaches take the column names as they are specified in the database. The problem with these approaches is that the column names displayed by `TableEdit` cannot be translated into other languages.
- The third approach to initializing the `TableEdit` element is flexible and allows, in addition to translation, setting the width of individual columns, horizontal alignment, and tooltips for each column:

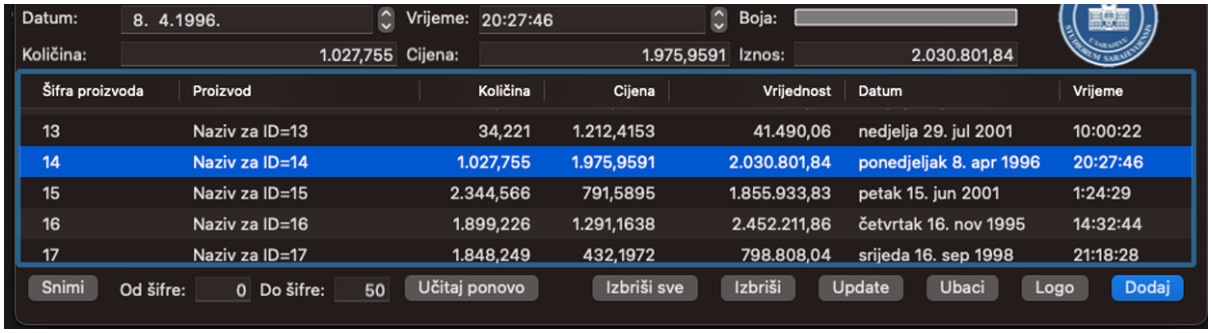
```
1  gui::Columns visCols(_table.allocBindColumns(8));
2  visCols << gui::ThSep::DoNotShowThSep
3          << gui::Header(0, tr("ItemID"), tr("ItemIDTT"))
4          << gui::Header(1, tr("ProductN"), tr("ProductNTT"), 170)
5          << gui::Header(2, tr("Quant"), tr("QuantTT"), 80, td::HAlignment::Right)
6          << gui::Header(3, tr("Price"), tr("PriceTT"), 80, td::HAlignment::Right)
7          << gui::Header(9, tr("Value"), tr("ValueTT"), 120, td::HAlignment::Right)
8          << gui::Header(4, tr("Date"), tr("DateTT"), 80)
9          << td::Time::Format::TimeOwnShortHMMSS << gui::Header(5, tr("Time"), tr("
       TimeTT"), 100)
10         << gui::Header(8, tr("Color"), tr("ColorTT"), 70);
11 _table.init(_pDS);
12
```

  - In this case, the `gui::Columns` container class is used, which consists of the parameters needed to define the visual properties of each column within the `TableEdit` header.
  - These visual parameters are stored in the `gui::Header` class, which is then passed to `gui::Columns`.
  - The first parameter in the `gui::Header` constructor is the column index within the `IDataSet`.

This initialization method allows detailed setting (format) of the display of each column. If the `TableEdit` columns were formatted as on the previous slide, the formatting of the date column would be identical to the one in the setting. If, for example, the date column were formatted as follows:

```
//...
<< td::Date::Format::DateWinLong << gui::Header(4, tr("Date"), tr("DateTT"), 150)
//...
```

This format would display the date with the day's name and the abbreviated month's name.



Slika 107: Example of a custom date format in TableEdit.

7.5 Handling Selection Changes

To react to changes in the selected row of a `gui::TableEdit` instance, it is necessary to implement the virtual method `onSelectionChanged` on the parent view. In our case, this implementation is shown in the code on the right.

```
bool onChangedSelection(gui::TableEdit* pTE) override
{
    if (pTE == &_table) {
        int iRow = _table.getFirstSelectedRow();
        if (iRow < 0) { // Table is empty, set to 0
            _id.toZero(); _name.toZero();
            _quant.toZero(); _price.toZero();
            _date.toZero(); _time.toZero();
            _color.setValue(td::ColorID::Black, false);
            return true;
        }

        dp::IDataSet* pDS = _table.getDataSet();
        auto& row = pDS->getRow(iRow);

        // Transfer data from IDataSet row to UI edit elements
        _id.setValue(row[0]);
        _name.setValue(row[1]);
        _quant.setValue(row[2]);
        _price.setValue(row[3]);
        _date.setValue(row[4]);
        _time.setValue(row[5]);

        td::UINT2 u2Val = row[8].wordVal();
        _color.setValue((td::ColorID)u2Val);

        _value.setValue(row[9]); // value
        return true;
    }
    return false;
}
```

- The `getFirstSelectedRow` method returns the index of the first selected row. If this number is less than 0, then `_table` is empty.
- After that, the data is taken from the corresponding `IDataSet`. So, `TableEdit` only provides information about the currently selected row.
- The `IDataSet` method `getRow` returns an array of `td::Variant` elements that are obtained from the stored values and values calculated by the delegate.
- After the `td::Variant` values are retrieved from the row, they are transferred to the corresponding editing elements by calling the `setValue` method.

7.6 Updating Values and Adding Rows

For simplicity, the operations that allow changing the content displayed by `TableEdit` will be explained in our example, which reacts to events after clicking on one of the buttons.

- All these events are handled in the virtual method `onClick(gui::Button*)`.
- The handling of a click on the "Reload" button is realized with:

```
// inside onClick
if (pBtn == &_btnReload)
{
    _paramFrom = _fromID.getValue().i4Val();
    _paramTo = _toID.getValue().i4Val();
    _table.reload();
    _table.selectRow(0, true);
    return true;
}

```

- Since `_table` has a pointer to the `IDataSet` interface, which is in turn bound to two parameters (`_paramFrom` and `_paramTo`), it is necessary to set the new values of these parameters before calling the `reload` method. After loading, it is set that the currently selected row is the row with index zero.
- This call will cause the `onChangedSelection` event to be generated (shown on the previous slide), which will transfer the data to the editing elements.

7.6.1 Modifying Table Content

Deleting the entire content displayed by `TableEdit` is achieved by calling the `clean` method:

```
if (pBtn == &_btnRemoveAll)
{
    _table.clean();
    return true;
}
```

- This method causes the entire content of the associated `IDataSet` container to be deleted.
- Other methods that edit elements require a call to `beginUpdate` before and `endUpdate` after editing one or more rows of `TableEdit`.

- For example, deleting the selected row after clicking the "Delete" button is realized as follows:

```
if (pBtn == &_btnDelete)
{
    int iRow = _table.getFirstSelectedRow();
    if (iRow < 0)
        return true;
    _table.beginUpdate();
    _table.removeRow(iRow);
    _table.endUpdate();
    return true;
}

```

Changing the content of the selected row after clicking the "Update" button is realized as follows:

```
if (pBtn == &_btnUpdate)
{
    int iRow = _table.getFirstSelectedRow();
    if (iRow < 0)
        return true;
    _table.beginUpdate();
    auto& row = _table.getCurrentRow();
    populateDSRow(row);
    _table.updateRow(iRow);
    _table.endUpdate();
    return true;
}
```

The data transfer itself from the editing elements to the `IDataSet` is done using the `populateDSRow` method, whose implementation is given on the right.

```
void populateDSRow(dp::IDataSet::Row& row)
{
    td::Variant val;
    _id.getValue(val); row[0].setValue(val);
    _name.getValue(val); row[1].setValue(val);
    _quant.getValue(val); row[2].setValue(val);
    _price.getValue(val); row[3].setValue(val);
    _date.getValue(val); row[4].setValue(val);
    _time.getValue(val); row[5].setValue(val);
    td::ColorID colorID = _color.getValue();
    td::WORD color = (td::WORD) colorID;
    row[8].setValue(color);
}
```

- In this case, calls to the `getValue` methods take the `td::Variant` values from the editing elements, which are then placed in the current row (`setValue`) obtained by calling `getCurrentRow` on the `IDataSet` instance.
- After filling the row elements, it is necessary to tell `TableEdit` that the data values have changed, which is achieved by calling `updateRow`.
- After that, `endUpdate` signals the end of the modification.

Inserting a new row before the currently selected row after clicking the "Insert" button is realized as follows:

```
if (pBtn == &_btnInsert)
{
    int iRow = _table.getFirstSelectedRow();
    if (iRow < 0)
        iRow = 0;
    _table.beginUpdate();
    auto& row = _table.getEmptyRow();
    populateDSRow(row);
    _table.insertRow(iRow);
    _table.endUpdate();
    return true;
}
```

- `TableEdit` generates an empty row by calling the `getEmptyRow` method, the row is filled using the local `populateDSRow` method, and then the `insertRow` method inserts the row at the specified position.
- All these operations are "enclosed" by `beginUpdate` and `endUpdate`.

Inserting (push_back) a new row at the very end of the `TableEdit` after clicking the "Add" button is realized as follows:

```
if (pBtn == &_btnPushBack) // In demo, this is "Dodaj" (Add)
{
    _table.beginUpdate();
    auto& row = _table.getEmptyRow();
    populateDSRow(row);
    _table.push_back();
    _table.endUpdate();
    return true;
}
```

7.7 Saving TableEdit Content to the Database

Saving the content of `TableEdit` after clicking the "Save" button is realized as follows:

```
if (pBtn == &_btnSave)
{
    saveData();
    return true;
}
```

The implementation of `saveData` is shown on the right.

```
bool saveData()
{
    dp::IStatementPtr pInsStat(_db->createStatement("INSERT INTO Test "
        "(ID, Name, Quant, Price, Date, Time, R4, R8, Color) "
        "VALUES(?,?,?,?,?,?,?,?,?)"));

    dp::Params parDS(pInsStat->allocParams());
    cnt::Array<td::Variant, 10> params;
    dp::RowWrapper row(params);

    if (!_pDS->initRowWrapper(row))
        return false;
```

```
13
14      //set string DB parameters
15      row[1].setDBTypeAndLen(td::nch, 64);
16      for (size_t i=0; i < parDS.size(); ++i)
17          parDS << row[i];
18
19      dp::Transaction tr(_db.ptr());
20      //delete some data
21      dp::IStatementPtr pDel(_db->createStatement("DELETE FROM Test"));
22      if (!pDel->execute())
23          return false;
24
25      size_t nRows = _pDS->getNumberOfRows();
26      for (size_t i=0; i < nRows; ++i)
27      {
28          _pDS->getRow(i, row);
29          if (!pInsStat->execute())
30              return false;
31      }
32
33      tr.commit();
34      return true;
35  }
```

- In this case, the parameters are placed in an array of `td::Variant` elements (L290).
- The `initRowWrapper` method initializes the types of columns stored in the `IDataSet` (L292).
- The column with index 1 requires setting the DB type and length (`setDBTypeAndLen`) L295.
- After that, the so-prepared array initializes the statement's parameters (L296,297).
- L300-302 delete the current content of the table.
- The transfer from the `IDataSet` and saving to the database is realized in L303-309.

7.8 natID Types

As is evident from all the examples so far, `natID` intensively uses its own classes for strings (`td::String`) as well as derived types for integer values (`td::WORD, td::INT2, td::UINT4, td::INT4, td::LINT8, td::LUINT8`). The question arises as to why standard C++ types for strings and standard types for integer values are not used.

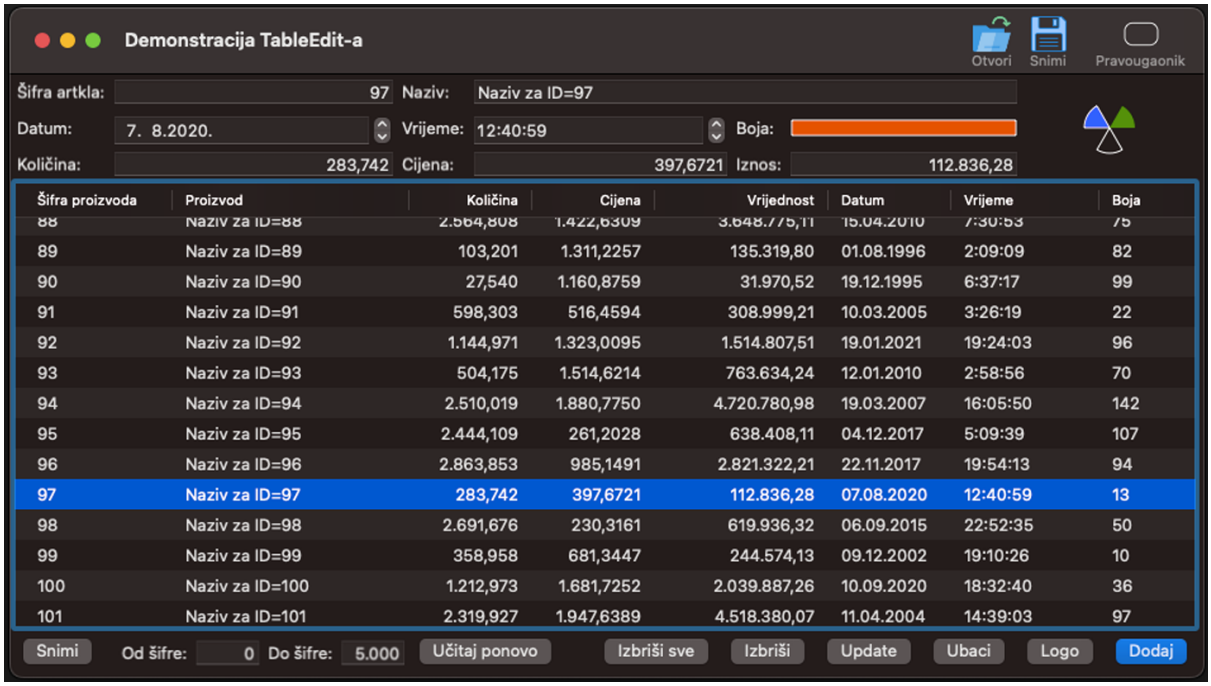
- Compared to the standard C++ `std::string`, `td::String` has a number of advantages and enables the realization of all interactions with the graphical interface and databases.
  - `td::String` uses "reference counting", which minimizes the number of string copies (allocation, copying, deallocation).
  - `td::String` supports Unicode and can be easily converted to any Unicode scheme required by the OS.
    - Standard C++ developed `std::wstring` for this purpose.
    - Unfortunately, `wchar_t` has a length of two bytes on Windows compilers and four bytes on Linux (Unix) compilers.

  - `td::String` can be easily initialized with `std::string` and `std::wstring`.

- As for integer values, their length in bytes is guaranteed, while the standard does not guarantee that, for example, an `int` is 4 bytes long.
  - `size_t`, for example, has a length determined by the type of compiler (the length is different if compiling with a 32-bit or 64-bit compiler).

- All of the above guarantees that binary data can be exchanged between different machines and different compilers with the highest possible performance.

7.9 Demonstration



Slika 108: Final demonstration of the TableEdit application.

8 2D Transformations

8.1 Point Representation

For a given 2D object, its transformation represents a change in its:

- Position (translation)
- Size (scaling)
- Orientation (rotation)

A column vector (2x1 matrix) can be used to represent a 2D point, specifically as x y.

The general form of a linear transformation can be written as:

$$x' = ax + by + c$$
$$y' = dx + ey + f$$

or in matrix form:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

8.2 Translation

Repositioning a point along a straight line.

- Initial point (x, y) and translation distance (t_x, t_y).
- New point (x', y').

$$x' = x + t_x$$
$$y' = y + t_y$$

- Or in matrix form $P' = P + T$, where:

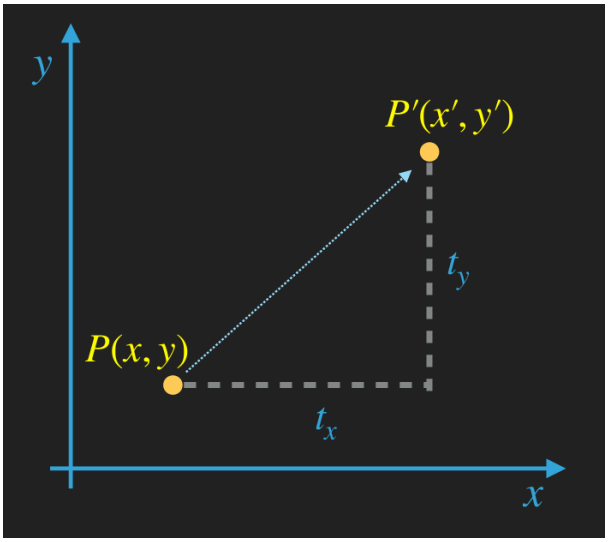
$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad P = \begin{bmatrix} x \\ y \end{bmatrix}, \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- Using a 2x1 vector:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- Using a 3x1 vector (homogeneous coordinates):

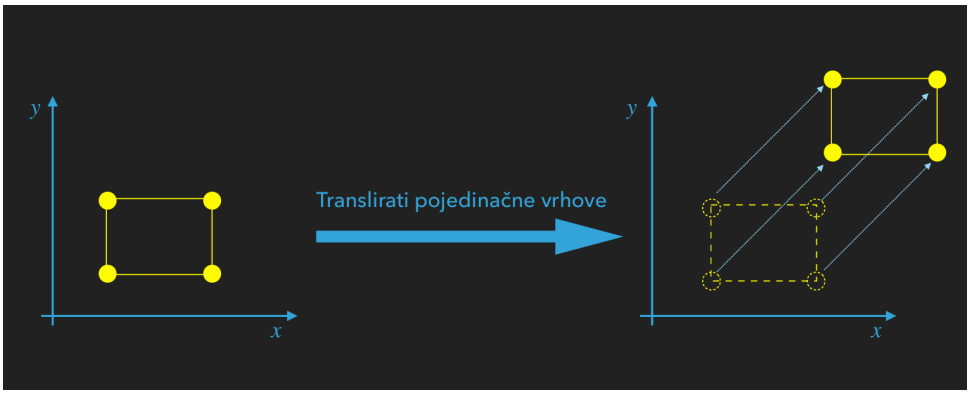
$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$



Slika 109: Translation of a point P to P'.

8.2.1 How to translate an object with multiple vertices?

To translate an object, translate its individual vertices.



Slika 110: Translating an object by translating each of its vertices.

8.3 Rotation

- Center of rotation: The origin of the coordinate system (0,0).
- How to calculate (x', y')?

$$x = r \cdot \cos(\phi), \quad y = r \cdot \sin(\phi)$$
$$x' = r \cdot \cos(\phi + \theta), \quad y' = r \cdot \sin(\phi + \theta)$$

  Using trigonometric identities:

$$x' = r \cdot \cos(\phi)\cos(\theta) - r \cdot \sin(\phi)\sin(\theta) = x \cdot \cos(\theta) - y \cdot \sin(\theta)$$
$$y' = r \cdot \sin(\phi)\cos(\theta) + r \cdot \cos(\phi)\sin(\theta) = y \cdot \cos(\theta) + x \cdot \sin(\theta)$$

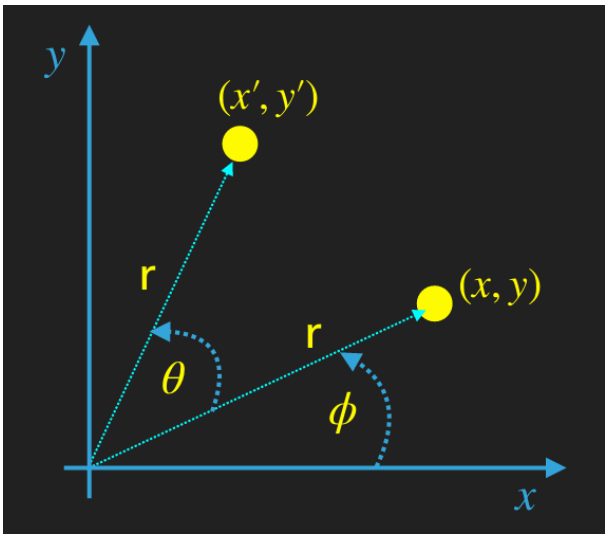
- In matrix form:

– 2x1 vector:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

– 3x1 vector:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$



Slika 111: Rotation of a point P around the origin.

8.3.1 Alternative Method: Complex Numbers

If we represent the point (x, y) in complex form, we have:

$$a = re^{j\phi}$$

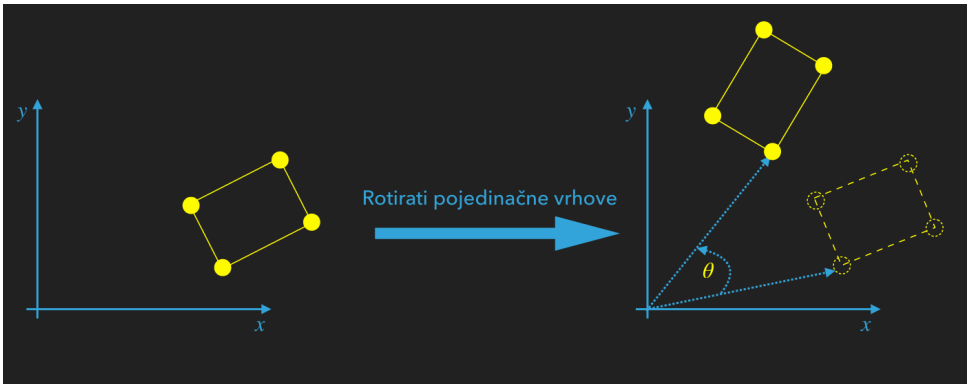
Q: What needs to be done to get the point (x', y') rotated by an angle θ around the origin of the coordinate system?

- The complex number must be multiplied by another complex number whose magnitude is $s = 1$ and angle is θ.
- $a' = a \cdot s$, or

$$a' = re^{j\phi} \cdot se^{j\theta} = rse^{j(\phi+\theta)} = re^{j(\phi+\theta)}$$

8.3.2 How to rotate an object with multiple vertices?

To rotate an object, rotate its individual vertices.



Slika 112: Rotating an object by rotating each of its vertices.

8.4 Scaling

Changing the size of an object using scaling factors (s_x, s_y).

$$x' = x \cdot s_x$$
$$y' = y \cdot s_y$$

In matrix form:

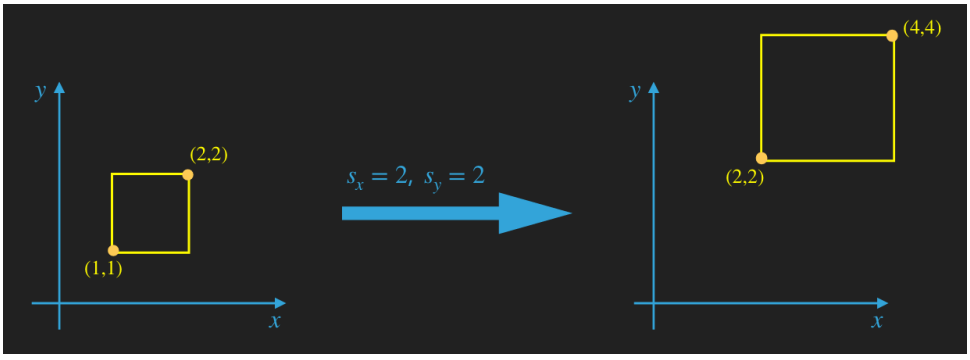
- 2x1 vector:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

- 3x1 vector:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Example: Scaling a rectangle with the bottom-left vertex at (1,1) and the top-right vertex at (2,2) with scaling factors $s_x = 2$ and $s_y = 2$.



Slika 113: Scaling a rectangle. Notice that not only the size changes, but also its position.

- Not only does the size of the object change, but it also moves.
- This is usually an undesirable effect.

8.5 Overview of All Transformations

8.5.1 2x2 Matrix Representations

- **Translation:** $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$
- **Rotation:** $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$
- **Scaling:** $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$

8.5.2 3x3 Matrix Representations

- **Translation:** $\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$
- **Rotation:** $\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$
- **Scaling:** $\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$

8.6 Why Use 3x3 Matrices?

- We can perform all transformations using matrix/vector multiplication.
- This allows all matrices to be pre-multiplied together.
- A point (x, y) should be represented as (x, y, w). Such points are called **homogeneous coordinates**.
- For 3D, these matrices are 4x4.

Q: Why is it important to have a graphics card?

- **A:** A graphics card is specialized for matrix (4x4) by vector (4x1) multiplication and has a very large number of specialized processing units that perform these operations in parallel.
- Since models can have a very large number of points, operations on the CPU would be very slow (modern CPUs have a maximum of about ten general-purpose units).

8.7 Center of Arbitrary Rotation

The standard rotation matrix is used for rotation around the coordinate origin (0,0). What if we want to rotate around an arbitrary center? For rotation around an arbitrary point $P(x_p, y_p)$ by θ:

1. Translate the object so that P coincides with the origin, creating the transformation matrix $T(-x_p, -y_p)$.

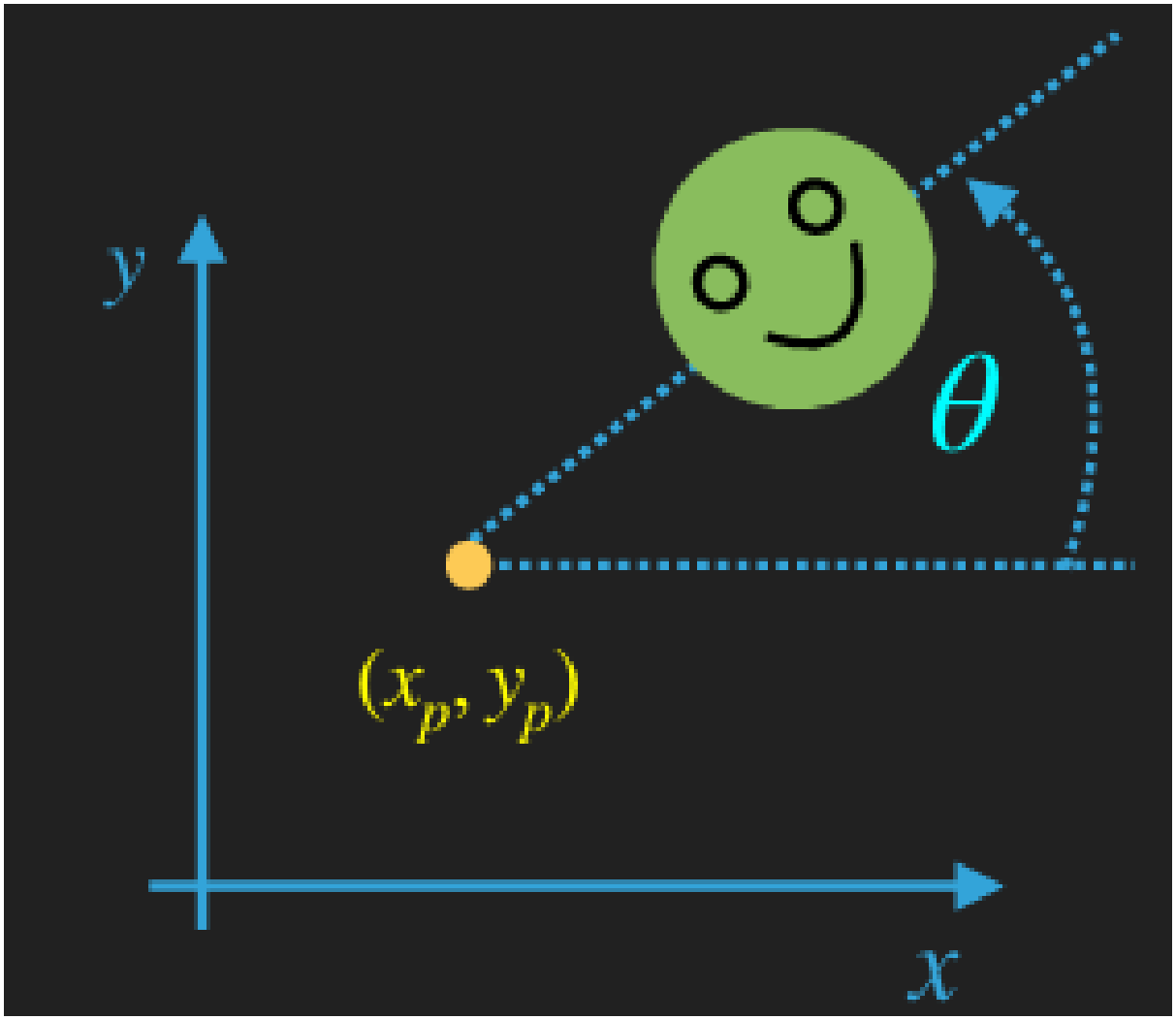
2. Rotate the object. This creates the transformation matrix $R(\theta)$.

3. Translate the object back. This step gives the final transformation matrix $T(x_p, y_p)$.

In matrix form, the complete transformation is:

$$T(x_p, y_p) \cdot R(\theta) \cdot T(-x_p, -y_p) \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_p \\ 0 & 1 & y_p \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_p \\ 0 & 1 & -y_p \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$



Slika 114: The three steps for rotating an object around an arbitrary point.

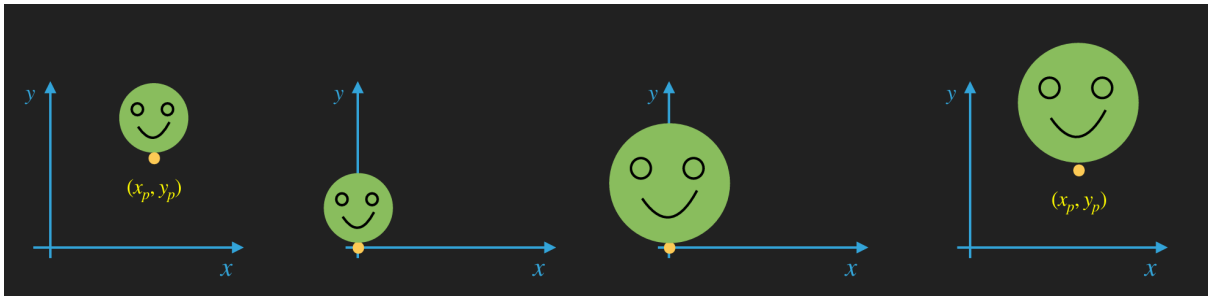
8.8 Arbitrary Point for Scaling

For scaling around an arbitrary central point $P(x_p, y_p)$:

1. Translate the object so that P coincides with the origin: $T(-x_p, -y_p)$.

2. Scale the object: $S(s_x, s_y)$.

3. Translate the object back: $T(x_p, y_p)$.



Slika 115: The three steps for scaling an object around an arbitrary point.

8.9 Affine Transformations

Translation, scaling, and rotation are all affine transformations. An affine transformation is:

- A transformed point $P'(x', y')$ is a linear combination of the original point $P(x, y)$, i.e.:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- Any 2D affine transformation can be decomposed into a rotation, followed by a scaling, and then a translation.
- **Affine matrix = translation x scaling x rotation**

8.10 Calculating the Matrix of a Complex Transformation

The process of applying several transformations sequentially to form a single total transformation. If a transformation of point P is applied using first matrix M_1, then transformation with M_2, and then M_3, we have:

$$(M_3 \cdot (M_2 \cdot (M_1 \cdot P))) = M_3 \cdot M_2 \cdot M_1 \cdot P$$

The composite matrix is $M = M_3 \cdot M_2 \cdot M_1$.

Therefore, because matrix multiplication is not a commutative operation, the first transformation that is performed on a point must be the last in the product of the composite transformation matrix.

8.11 The Order of Transformations is Important!

- Matrix multiplication is associative:

$$M_3 \cdot M_2 \cdot M_1 \cdot P = (M_3 \cdot M_2) \cdot M_1 = M_3 \cdot (M_2 \cdot M_1)$$

- Transformation products may not be commutative:

$$A \cdot B \neq B \cdot A$$

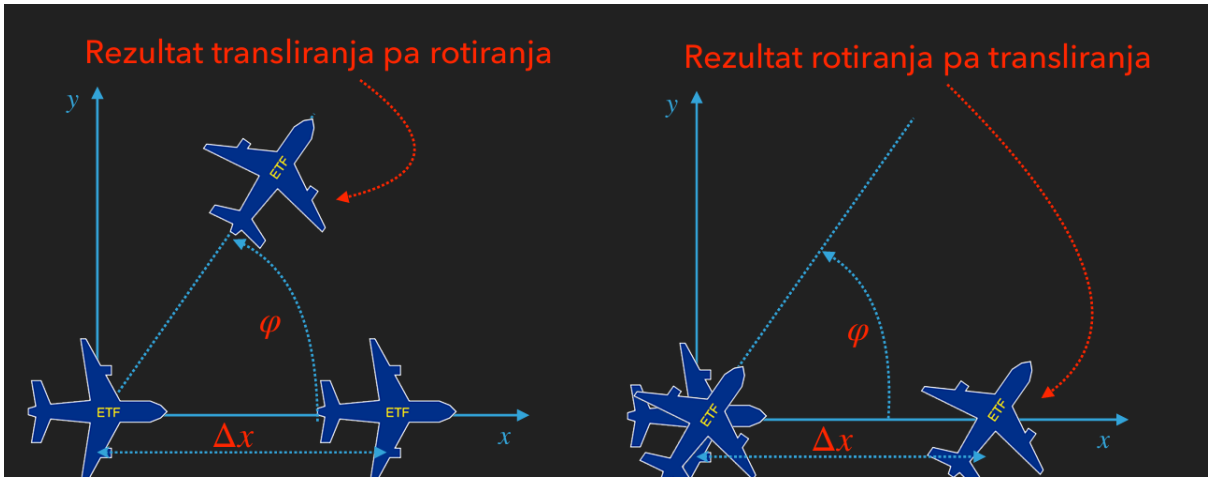
- Some cases where $A \cdot B = B \cdot A$:

A	B
Translation	Translation
Scaling	Scaling
Rotation	Rotation
Uniform scaling ($s_x = s_y$)	Rotation

Q: Can we prove this claim?

8.11.1 Rotation and Translation are Not Commutative

Example: Let's translate an airplane model by Δx and then rotate it by an angle ϕ, and then repeat this in the reverse order.



Slika 116: The result of translate then rotate (left) is different from rotate then translate (right).

9 Visualization of Large Models (Canvas, Scroll)

9.1 2D Graphics: Canvas

- `gui::Canvas` is a class that allows the drawing of elementary graphical shapes, images, symbols, and text.
- For the graphical visualization (drawing) of any element, it is necessary to create a new class inherited from `gui::Canvas`.
- This class prepares the graphics context and, when needed, calls the virtual method `onDraw`:

```
virtual void onDraw(const gui::Rect& rect);

```

  - The `rect` parameter provides information about which part of the Canvas needs to be redrawn (it does not have to be the entire Canvas).
- The following rules must be followed when drawing on a Canvas:
  - The complete code for drawing must be called from within the `onDraw` method.
  - The `onDraw` method should never be called directly (the system calls it after preparing the graphics context).
  - To initiate a new call to `onDraw`, the `reDraw` function must be used.
    - This function is asynchronous and it places information for the system to prepare the context and call the `onDraw` method at some future possible moment.

In addition to the ability to create 2D graphical content, the `Canvas` class adds a set of virtual methods for interaction with it.

- For mouse/touch clicks:

```
virtual void onPrimaryButtonPressed(const gui::InputDevice& inputDevice);
virtual void onPrimaryButtonReleased(const gui::InputDevice& inputDevice);
virtual void onSecondaryButtonPressed(const gui::InputDevice& inputDevice);
virtual void onSecondaryButtonReleased(const gui::InputDevice& inputDevice);
virtual void onPrimaryButtonDblClick(const gui::InputDevice& inputDevice);

```

- For zoom management:

```
virtual bool onZoom(const gui::InputDevice& inputDevice);

```

- For keyboard interaction:

```
virtual bool onKeyPressed(const gui::Key& key);
virtual bool onKeyReleased(const gui::Key& key);

```

- For detecting changes in the Canvas dimensions:

```
virtual void onResize(const gui::Size& newSize);

```

- For these methods to be called, they need to be enabled in the Canvas's constructor (not all are always needed).

9.2 Transformation

- The `gui::Transformation` class allows for the simple creation of transformation matrices using the `translate`, `rotate`, and `scale` methods.
- The resulting transformation matrix can be set to be the current matrix in the context by calling the method:

```
void setToContext() const;

```

  - In this case, the previous composite transformation matrix that was in the context is replaced with the new one.

- If you want to multiply the current value of the transformation matrix in the context from the right side with the created transformation, the method is used:

```
void appendToContext() const;

```

- Both previous cases lose information about the composite transformation matrix before these activities. It is often necessary to save the current value of the transformation matrix in the context and restore it after performing changes and drawing a model that requires additional transformations. This can be achieved using a pair of methods:

```
static void saveContext();
static void restoreContext();

```

- It is possible to call `saveContext` multiple times in a row. It is important that each `saveContext` is followed by one `restoreContext`.

9.3 Shape

The `gui::Shape` class enables the drawing of graphical primitives:

- line
- rectangle
- circle
- arcs
- parts of a circle (pie)

- polygon
- Bezier multi-segment curve (`Shape::Bezier` class)
- ...

Creating the appropriate shape is achieved by calling the corresponding method:

```
void createLines(const gui::Point* points, size_t nPoints, ...);
void createPolyLine(const gui::Point* points, size_t nPoints, ...);
void createPolygon(const gui::Point* points, size_t nPoints, ...);
void createRect(const gui::Rect& rect, ...);
void createRoundedRect(const gui::Rect& rect, CoordType radius, ...);
void createOval(const gui::Rect& rect, ...);
void createArc(const gui::Circle& circle, float fromAngle, float toAngle, ...);
void createPie(const gui::Circle& circle, float fromAngle, float toAngle, ...);
void createCircle(const gui::Circle& circle, ...);
void createCircles(const gui::Circle* circles, size_t nCircles, ...);
```

- The creation of elements should be done in scene preparation and not within the `onDraw` method.
- The drawing of elements itself (within `onDraw`) is performed with one of the following calls:

```
void drawWire(td::ColorID lineColor) const;
void drawWire(td::ColorID lineColor, float lineWidth) const;
void drawFill(td::ColorID fillColor) const;
void drawFillAndWire(td::ColorID fillColor, td::ColorID lineColor) const;
void drawFillAndWire(td::ColorID fillColor, td::ColorID lineColor,
                     float lineWidth) const;

```

9.4 Introduction to Bézier Curves

- Bézier curves are parametric curves that are often used in computer graphics and modeling.
- Pierre Bézier developed them in the 1960s for car design at Renault.
  - Independently developed by Paul de Casteljau at Citroën.
- Curves are defined by a parameter t that ranges from 0 to 1 and control points $P_0, P_1, ..., P_n$.
- The mathematical formulation of a Bézier curve is based on the use of Bernstein polynomials:

$$B(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i, \quad t \in [0,1]$$

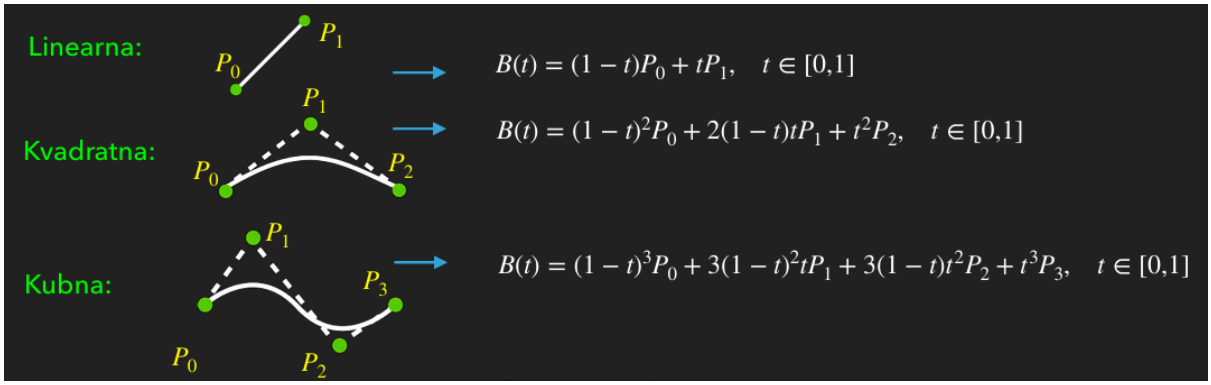
  where $B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$ is the Bernstein polynomial of degree n.
- De Casteljau developed a recursive algorithm for evaluating Bézier curves:

$$P_{i,j}(t) = (1-t) P_{i,j-1} + t P_{i+1,j-1}$$

9.4.1 Properties of Bézier Curves

- **Continuity:**
  - C^0 continuity: The start and end points are connected.
  - C^1 continuity: The tangent is smooth at the join.
  - C^2 continuity: The curvature is smooth at the join.
- **Generalization:**
  - It is possible to define Bézier surfaces for 3D modeling.
- **Limitations:**
  - Complex shapes require a large number of control points.
- **Today:** A standard in the graphics industry and a basic building block in modern graphics tools.

9.4.2 Bézier Curves of the First, Second, and Third Order



Slika 117: Linear, Quadratic, and Cubic Bézier curves.

- **Bezier support in natID:**

```
Bezier& moveTo(const Point& p);
Bezier& lineTo(const Point& p, bool closePath = false);
Bezier& quadraticTo(const Point& p, const Point& ctrlPoint, ...);
Bezier& cubicTo(const Point& p, const Point& ctrlPoint1,
                const Point& ctrlPoint2, bool closePath);

```

- The `Shape` class supports creating Bézier curves from multiple segments.
  - Segments can be connected using the `lineTo`, `quadraticTo`, and `cubicTo` methods.
  - In addition, segments can have breaks (starting from another point), which is achieved using the `moveTo` method.
- For details, see the `testCanvas` project and the `ViewCanvas.h` class within it.

9.4.3 Simulating Complex Curves with Bézier Segments

A complex curve of a higher order can be approximated using multiple Bézier curves of the second (quadratic) and third (cubic) order. The key is to ensure C^0, C^1, and C^2 continuity between segments so that the transitions are smooth.

Let Q be the points of the next segment and P be the points of the previous segment. To ensure C^0, C^1, and C^2 continuity, the following must hold:

- Positional continuity C^0: $P_n = Q_0$
- Tangential continuity C^1: $P_{n-1} - P_n = Q_1 - Q_0$
- Curvature continuity C^2: $\frac{P_{n-2}-P_{n-1}}{P_{n-1}-P_n} = \frac{Q_0-Q_1}{Q_1-Q_2}$

From this, we conclude that to achieve continuity, it is necessary to use third-order Bézier segments.

9.4.4 Example of Using Bézier Curves in natID

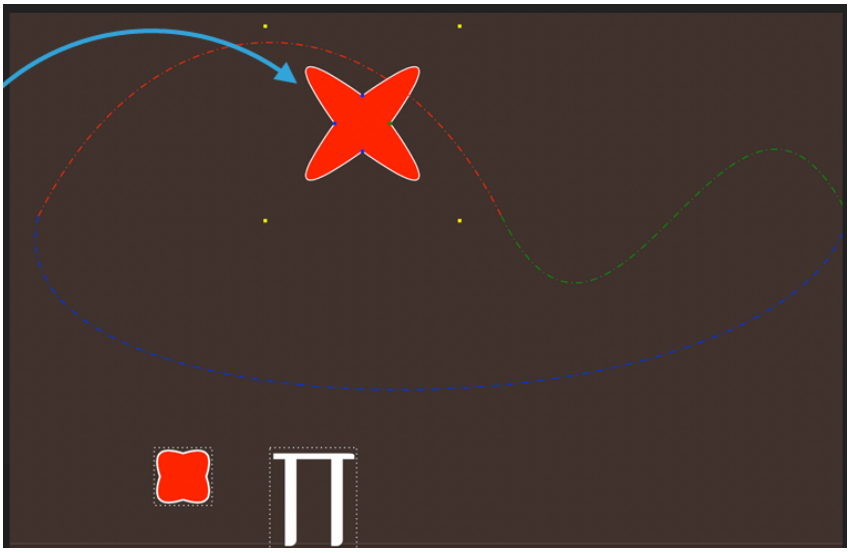
The complete implementation is shown in the `testCanvas` project in the `ViewCanvas.h` file. `ViewCanvas` defines the following `Shape` attributes:

```
gui::Shape _shapeBezier;
gui::Shape _lineBez1;
gui::Shape _lineBez2;
gui::Shape _lineBez3;
```

In the constructor, Bezier Shape objects are created:

```
// 4-wings star
_shapeBezier.createBezier().moveTo({rB+cX,cY}) //P1
    .quadraticTo({cX, rB+cY}, {rB+cX+dCtrl, rB+cY+dCtrl}) //P2, P2d
    .quadraticTo({-rB+cX, cY}, {-rB+cX-dCtrl, rB+cY+dCtrl}) //P3, P3c
    .quadraticTo({cX, -rB+cY}, {-rB+cX-dCtrl, -rB+cY-dCtrl}) //P4, P4c
    .quadraticTo({rB+cX, cY}, {rB+cX+dCtrl, -rB+cY-dCtrl}, true); //P1, P1c

// simple lines
_lineBez1.createBezier(1.0, td::LinePattern::DashDot).moveTo({100, 300})
    .cubicTo({500,300}, {200,100}, {400,100});
_lineBez2.createBezier(1.0, td::LinePattern::DashDot).moveTo({500,300})
    .cubicTo({800,300}, {600,500}, {700,100});
_lineBez3.createBezier(1.0, td::LinePattern::DashDot).moveTo({100,300})
    .cubicTo({800,300}, {50,500}, {750,500});
```

Drawing is done with `Shape` methods just like for any other `Shape` object (`drawWire`, `drawFill`, ...).



Slika 118: Example of a complex shape drawn using Bézier curves.

9.5 Image and Symbol

- **Images** are usually created as part of the resources but can also be loaded directly from a file. For performance reasons, loading should never be within the `onDraw` method. The drawing of the image itself is executed by calling the `draw` method:

```
void draw(const gui::Rect& rect, AspectRatio aspectRatio = AspectRatio::Keep,
          td::HAlignment hAlign=td::HAlignment::Center,
          td::VAlignment vAlign=td::VAlignment::Center) const;

```

- **Symbols** (`gui::Symbol`) are a class that allows grouping multiple graphical primitives into an XML file. Symbols are vector graphics and display high-quality content regardless of resolution. They are usually used as resources but can also be loaded directly. Drawing is done with methods like:

```
void draw(gui::CoordType dx, gui::CoordType dy, float rotAngleRad=0, ...);
void drawInRect(const gui::Rect& boundRect, float rotationRad=0, ...);

```

- All set transformations apply to all elements (including images and symbols). Therefore, images can be rotated, scaled, etc.

9.6 DrawableString and Scrolling

- **`gui::DrawableString`** is used for displaying text on a `gui::Canvas`. For performance, the string should be initialized outside the `onDraw` method. The drawing itself within `onDraw` can be done by calling one of two methods:

```
void draw(const gui::Rect& r, gui::Font::ID fntID, ...);
void draw(const gui::Point& r, gui::Font::ID fntID, ...);

```

  The current transformation matrix will be applied to the text. In this way, the text can be displayed rotated, vertical, or oriented as desired, just like any other graphical element.

- **Scrolling (`gui::ViewScroller`):** Sometimes the content being drawn is too large to fit in the visible part of the Canvas. The `gui::ViewScroller` class allows viewing parts of the model, i.e., the parts being drawn on the Canvas. In addition, this class allows `zoomToPoint`. Setting the view for scrolling (Canvas or some other) is done by calling the `setContentView` method:

```
1 void setContentView(BaseView* pContentView);
2
```

9.7 Splitter Layout

- The layouts used in the previous part of the lectures did not have the possibility of user adjustment.
- The `gui::SplitterLayout` class divides the assigned space horizontally or vertically into two parts.
- The user can regulate the dimensions of both parts.
- One of these two parts is declared as auxiliary and can be minimized, thus leaving maximum space for the main view.

10 Introduction to Real-Time Systems

10.1 Real-Time Systems

- **Real-time computing (RTC)**, or reactive computing, is a term used in computer science for hardware and software systems that are subject to a "real-time constraint," for example, from an external event to a system response.
- Programs in real-time must guarantee a response within specific time constraints, often called "deadlines."
- A system is said to operate in real-time if the total correctness of a system's operation depends not only on its logical correctness but also on the time in which it is performed.
- Real-time systems, as well as their deadlines, are classified according to the consequences of missing (not meeting) a deadline:
  - **Soft RT** - the usefulness of the result degrades after the deadline expires, which degrades the quality of the system's service.
  - **Firm RT** - rare deadline misses are tolerable but may degrade the system's quality of service. The usefulness of the result is zero after the deadline expires.
  - **Hard RT** - failure to meet any deadline is a total system failure.

Real-time computing is sometimes misunderstood as high-performance computing, but this is not an accurate classification.

- For example, a massive supercomputer performing a simulation can offer impressive performance but does not perform real-time computation.
  - Conversely, once the hardware and software for, say, an Anti-lock Braking System (ABS) are designed to meet their specified deadlines, further performance improvement is not mandatory.
- The most important requirement of a real-time system is a consistent output, not high throughput.
- Some types of software, like many chess-playing programs, can fall into both categories. For example, a chess program designed to play in a tournament with a clock will have to decide on a move before a certain deadline or lose the game, so that is real-time computing. In another case, when the software is used for analysis and learning, the computation does not have to be done in real-time.
- High performance is indicative of the amount of processing performed in a given amount of time, while real-time is the ability to complete processing to produce a useful output in the available time.

10.2 Real-Time Operating System (RTOS)

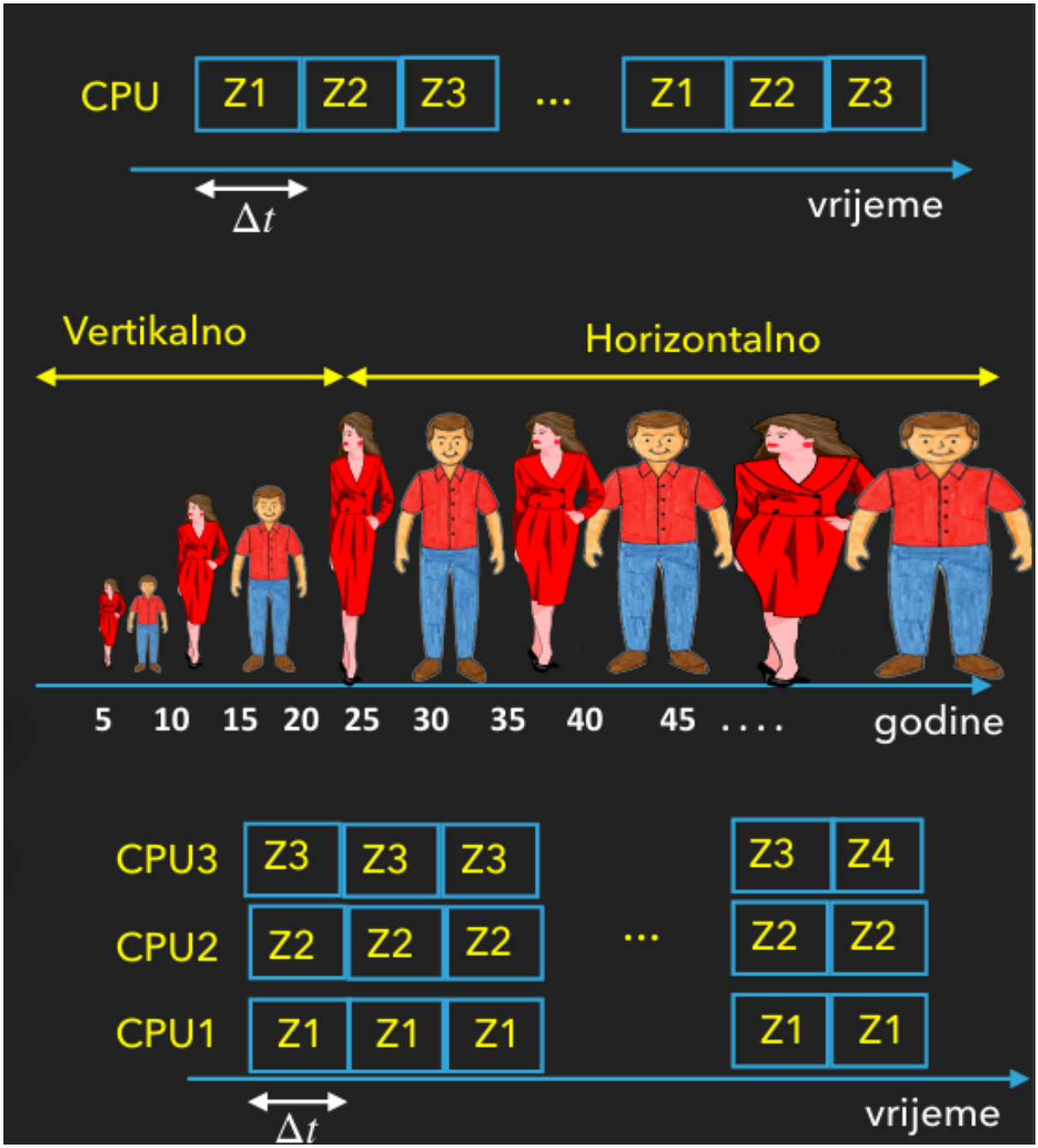
A Real-Time Operating System (RTOS) is an operating system (OS) intended to serve real-time applications that process data as it comes in, usually without buffer delays.

- An RTOS executes multiple tasks with different priorities.
- The most commonly used designs regarding task execution are:

- **Event-driven:** switches tasks only when an event that requires a higher-priority task to run occurs. After the higher-priority task is finished, it resumes the previously started task. (preemptive priority scheduling).
- **Time-sharing:** switches tasks at a regular time interval (round-robin).

10.3 Improving Computer Power

- In the past, embedded systems often used single-processor systems where tasks executed in virtual parallelism using the principle of time-sharing.
- During that period, improvements in processor power were mainly achieved by adding new instructions and increasing frequency.
- Over time, due to physical limitations, the increase in frequency was replaced by an increase in the number of processor cores on a common processor die.
- This led to true parallelism in the execution of different tasks.



Slika 119: Evolution from single-core time-sharing to multi-core true parallelism.

10.4 Concurrent (Parallel) Code Execution

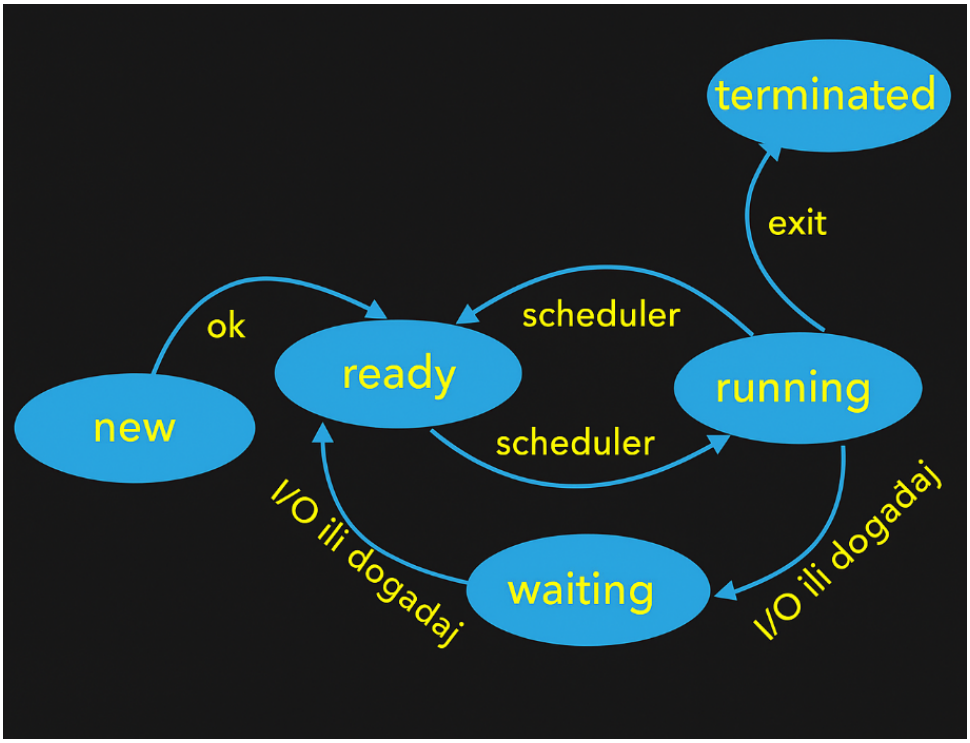
Concurrency in the execution of tasks (code) can be achieved in two ways:

- using processes (eng. job, process)
- using threads (eng. thread)

A process is a program that is being executed and can be in the following states:

- **new** - the process is just being created

- **ready** - the process is ready to be executed
  - waiting for the scheduler (time slot)
- **running** - the process instructions are currently being executed in the processor
- **waiting** - the process instructions are not being executed
  - the process is waiting for some external event
- **terminated** - the process has finished execution

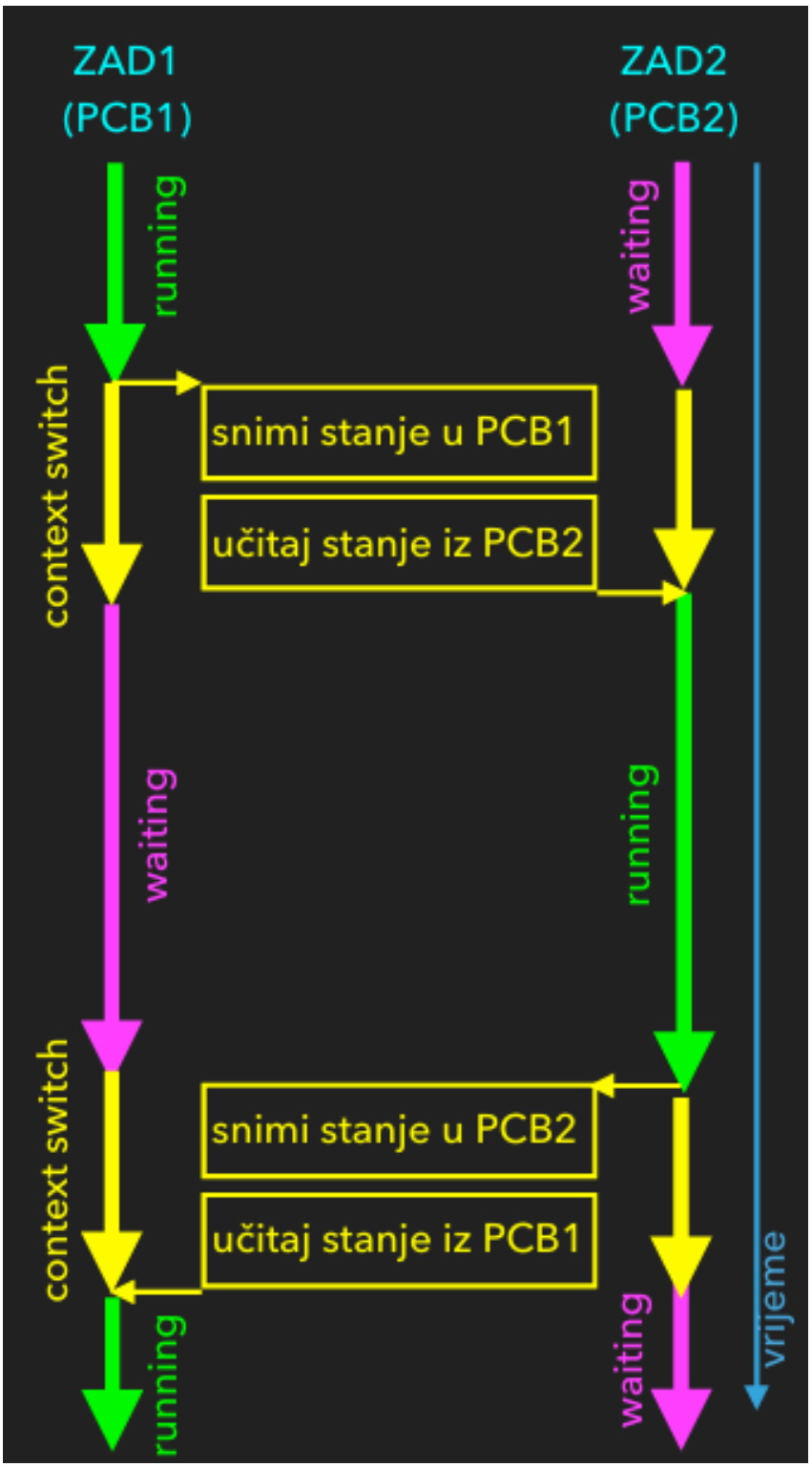


Slika 120: Diagram of process states.

10.5 Process Control Block (PCB) and Context Switch

- When changing tasks that are executing on the processor, the RTOS must save the complete state of the process (task) that is interrupting its execution, and restore the state of the process (task) that is starting to execute its instructions (context switch).
- This procedure includes saving/loading:
  - Processor state
  - Program counter
  - CPU registers
  - Memory management information
  - I/O status information
    - open files, network sockets, pipes,...
  - ...

- During the task change, the CPU is not doing anything useful, so achieving parallelism using processes is a costly procedure.



Slika 121: Diagram illustrating a context switch between two processes.

10.6 Programmatic Starting of Processes

POSIX (linux, solaris, macos,...)

```c
int main() {
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process
    */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

Windows

```c
void main(int argc, char *argv[]) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (!CreateProcess(NULL, argv[1],
        NULL, NULL, FALSE, 0, NULL,
        NULL, &si, &pi)) {
        return;
    }
    // Wait until child process exits
    WaitForSingleObject(pi.hProcess,
                        INFINITE);
    // Close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

10.7 Threads

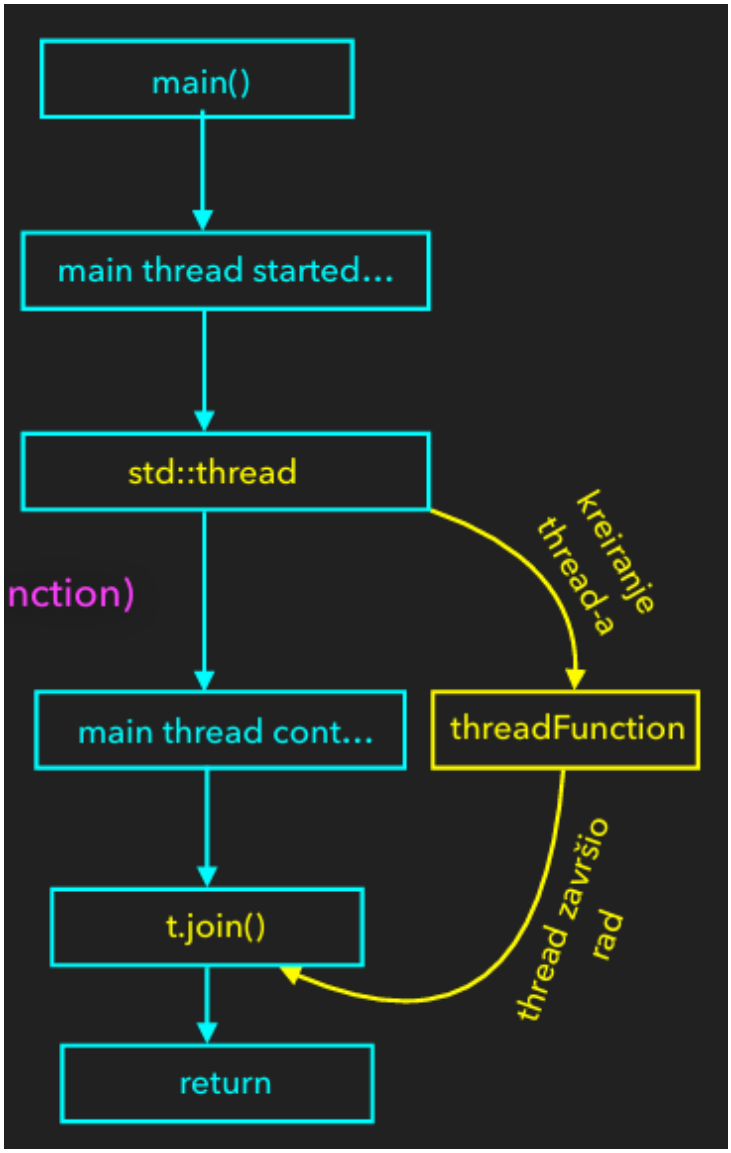
- Since changing processes requires a fairly expensive context switching process, using processes to achieve parallelism (multitasking) reduces system performance.
- A thread can be considered a lightweight process in terms of parallelism.
- A thread shares the memory space of its process, so a context switch requires much less data.
- A single process can have multiple threads.
- Much easier communication (data exchange) between threads.
  - processes must use inter-process communication (pipes, sockets, shared memory,...)
  - threads execute in the same memory space
  - there are no communication costs between threads.
- C++11 provides fully standardized support for working with threads, including synchronization.
  - There is no need for OS-specific code.

10.8 Starting a Thread (C++11)

```c
#include <iostream>
#include <thread>

void threadFunction() {
    // place your code here
    std::cout << "thread function\n";
}

```

```cpp
int main() {
    std::cout << "main thread started...\n";
    // thread starts immediately
    std::thread t(&threadFunction);
    std::cout << "main thread cont...\n";

    // functionality that executes in main thread

    // main thread waits for thread t to finish
    t.join();
    return 0;
}
```

To compile (e.g., Linux system): `$ g++ t1.cpp -o t1 -std=c++11 -pthread`



Slika 122: Flowchart for C++11 thread creation and execution.

10.9 Using a Class Method as a Thread Function (C++11)

There are several ways a method from a class can be started as a thread. The three most common ways are:

- implementing a function object (functor - method `void operator()()`)
- implementing a "regular" method that is called from the `std::thread` class constructor
- using `std::bind`

Functor Example

```
#include <iostream>
#include <thread>
class MyClass {
public:
    void operator()() {
        std::cout << "thread functor\n";
    }
};

int main(){
    MyClass fnctor;
    std::thread t(&fnctor);
    std::cout << "main thread\n";
    t.join();
    return 0;
}
```

Member Function Example

```
#include <iostream>
#include <thread>
class MyClass {
public:
    void threadFn() {
        std::cout << "threadFn\n";
    }
};

int main(){
    MyClass myClass;
    std::thread t(&MyClass::threadFn,
                  &myClass);
    std::cout << "main thread\n";
    t.join();
    return 0;
}
```

10.10 Passing Parameters to a Thread

Passing parameters to a thread is done by adding parameter variables to the `std::thread` class constructor. In the case of passing parameters by reference, it is necessary to use the `std::ref` function within the thread constructor.

Pass by Value

```cpp
#include <iostream>
#include <thread>
#include <string>

void threadFunction(std::string s){
    std::cout << "threadFunction ";
    std::cout << "msg is = " << s;
    std::cout << std::endl;
}

int main(){
    std::string s = "This is a param!";
    std::thread t(&threadFunction, s);
    std::cout << "main thread cont...\n";
    t.join();
    return 0;
}
```

Pass by Reference

```cpp
#include <iostream>
#include <thread>
class MyClass {
public:
    void thFn(int& x) {
        std::cout << "threadFn\n";
        x += 10;
    }
};

int main(){
    MyClass* pMC = new MyClass;
    int val = 5;
    std::thread t(&MyClass::thFn, pMC,
                  std::ref(val));
    t.join();
    std::cout << "main thread: " << val;
    std::cout << std::endl;
    delete pMC;
    return 0;
}
```

10.11 Shared Resources and Synchronization

- Sometimes, two or more threads may require changes to a shared resource (variable), which can lead to inconsistency. For example, if two threads write messages to the console, the console might display "mixed" messages.
- To guarantee that only one thread has access to a shared resource, it is necessary to use the `std::mutex` class and its `lock` and `unlock` methods.

10.11.1 Race Condition Example

Shared resource: `std::cout`.

```cpp
#include <iostream>
#include <thread>

void threadFunction(){
    for (int i = -100; i < 0; i++)
        std::cout << "thread function: " << i << "\n";
}

int main(){
    std::thread t(&threadFunction);
    for (int i = 0; i < 100; i++)
        std::cout << "main thread: " << i << "\n";
    t.join();
    return 0;
}
```

Possible interleaved output:

```
thread function: main thread: 0
```

```
main thread: 1
main thread: 2
-100
thread function: -99
thread function: -98
...
thread function: -82main thread: 3
main thread: 4
...
```

10.11.2 Synchronization with Mutex

```cpp
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mu;

void sharedCout(const char* msg, int id) {
    mu.lock();
    std::cout << msg << ":" << id << std::endl;
    mu.unlock();
}

void threadFunction() {
    for (int i = -100; i < 0; i++)
        sharedCout("thread function", i);
}

int main(){
    std::thread t(&threadFunction);
    for (int i = 100; i > 0; i--)
        sharedCout("main thread", i);
    t.join();
    return 0;
}
```

10.12 Mutex and Deadlock

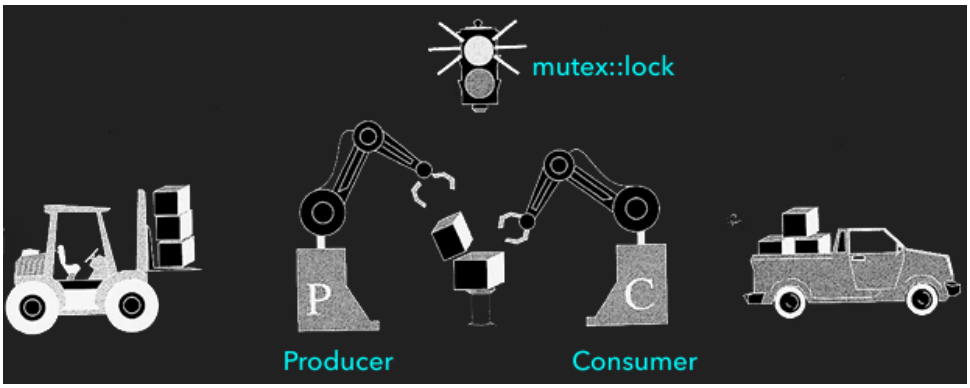
- **Mutex** comes from the first letters of the English words **mutually exclusive**.
- If multiple threads require a lock on a mutex, only one thread will gain access while the other threads will be put into a "waiting" status (they will not consume processor time).
- After the thread that gained access via the `lock` method finishes its work on the shared resource, it is necessary to perform an `unlock` as soon as possible.
- After an `unlock`, one of the threads that were in the "waiting" status will be allowed to work on the shared resource.
- This guarantees that only one thread at a time has access to the shared resource.

- It is very important to ensure that for every `lock`, an `unlock` is called after working with the shared resource; otherwise, a **deadlock** can occur.
  - In that case, the entire work of the threads that require the shared resource becomes blocked, i.e., they are in a "waiting" status, which can ultimately lead to the blocking of the entire process.

10.13 Signaling Between Threads

The question arises whether a mutex is sufficient for synchronization between two or more threads.

- To illustrate the problem, let's consider the image where a delivery person unloads boxes, and a robotic arm (producer) places the boxes in a common area (shared resource). From the other side, another robotic arm (consumer) takes the placed box, performs certain operations on it, and places it in a vehicle for transport.
- Let's say the vehicles that deliver and take away the boxes operate completely asynchronously and at different speeds.
- For optimal operation, it is necessary to ensure that:
  - there can be a maximum of one box in the shared space,
  - neither of the robotic arms makes a futile attempt (e.g., the consumer moves to take a box that is not available).
- We conclude that using only a mutex is not possible to find an optimal solution to this problem.
- In computer terminology, this problem is called the **Producer-Consumer problem**. **Q: What needs to be changed?**

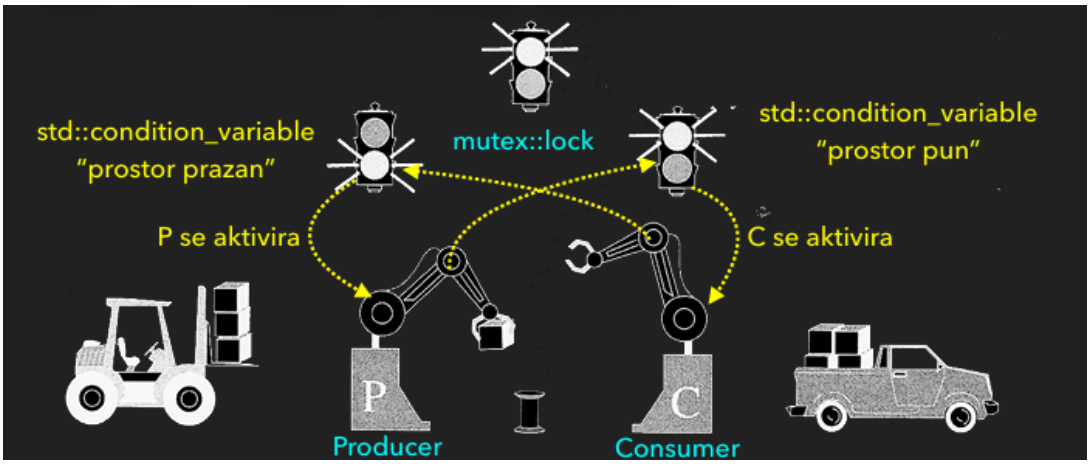


Slika 123: The producer-consumer problem with only a mutex.

We conclude that for the optimal operation of such a system, "space empty" or "space full" signals should be introduced:

- The producer P would signal to the consumer C that the "space is full" after delivering a box.
- The consumer C would become active after receiving the "space is full" signal.
- The consumer C would signal to the producer P that the "space is empty" after taking the box.
- The producer P would become active after receiving the "space is empty" signal.

This is programmatically possible using the `std::condition_variable` class (C++11).



Slika 124: Producer-consumer solution with condition variables for signaling.

10.14 Bounded-Buffer Problem

The producer-consumer problem is also known as the bounded-buffer (circular buffer) problem.

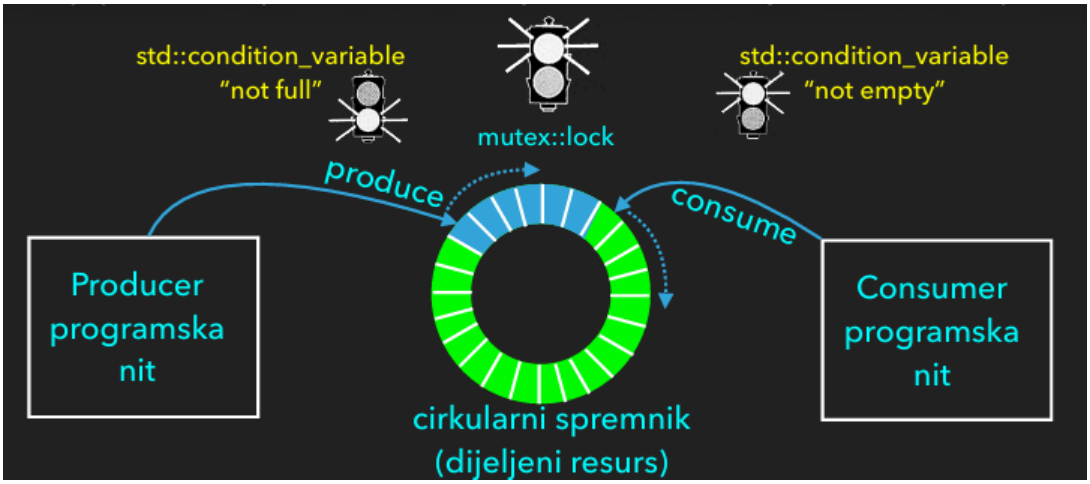
- A bounded buffer is a memory array of length n.
- The producer thread writes to the buffer at the first free position.
  - It must not write more than n locations; otherwise, data would be lost.
  - If the end of the buffer is reached, the write pointer must be moved to the starting position (i.e., after position $n - 1$ comes position 0).
- The consumer thread reads data from the buffer from the earliest written location.
  - It must not attempt to read if there is no data in the buffer.
  - If the end of the buffer is reached, the read pointer must be moved to the starting position.

Slika 125: Circular buffer diagram for the Bounded-Buffer problem.

The algorithm for solving the bounded-buffer problem can be described by observing the previously defined procedures:

- The producer waits for a "not full" signal and then writes data to the bounded buffer after receiving the signal.

- Since there can be multiple producers or consumers, the producer thread must call a mutex lock before any operation on the shared resource (the circular buffer).
- After writing data to the circular buffer, the producer generates (sets) a "not empty" signal, thereby letting the consumer(s) know that there is data in the buffer.
- The consumer waits for a "not empty" signal, and after receiving that signal, it locks the mutex and accesses the data from the buffer.
- After reading data from the buffer, the consumer generates a "not full" signal, thereby "waking up" producers who are waiting.



Slika 126: Algorithm flow for the Bounded-Buffer problem.

10.14.1 Bounded-Buffer Problem - Pseudo-code

Producer

```
// Shared resources (globals)
const int g_capacity = 100;
int g_buffer[capacity];
int g_front, g_rear;
std::mutex g_lock;
std::condition_variable not_full;
std::condition_variable not_empty;

// Producer
do {
    ...
    // create element to write
    ...
    wait_for_signal(not_full);
    lock(g_lock);
    ...
    // check index
    ++g_front;
    if (g_front >= g_capacity)
        g_front = 0;
    // write element to g_front
    g_buffer[g_front] = ...
    release(mutex);
    signal(not_empty);
} while (true);
```

Consumer

```
// Shared resources (globals)
...

// Consumer
do {
    ...
    wait_for_signal(not_empty);
    lock(g_lock);
    ...
    // check index
    ++g_rear;
    if (g_rear >= g_capacity)
        g_rear = 0;
    // read element from g_rear
    ... = g_buffer[g_rear];
    release(mutex);
    signal(not_full);
} while (true);
```

10.14.2 Bounded-Buffer Problem - C++11 Implementation

```
struct BoundedBuffer {
    int* buffer;
    int capacity, front, rear, count;

    std::mutex lock;
    std::condition_variable not_full;
    std::condition_variable not_empty;

    BoundedBuffer(int capacity)
    : capacity(capacity), front(0), rear(0), count(0) {
        buffer = new int[capacity];
    }

    ~BoundedBuffer(){ delete[] buffer; }

    void put(int data){
        std::unique_lock<std::mutex> l(lock);
        not_full.wait(l, [this](){return count != capacity; });
        buffer[rear] = data;
        rear = (rear + 1) % capacity;
        ++count;
        l.unlock();
        not_empty.notify_one();
    }

    int get(){
```

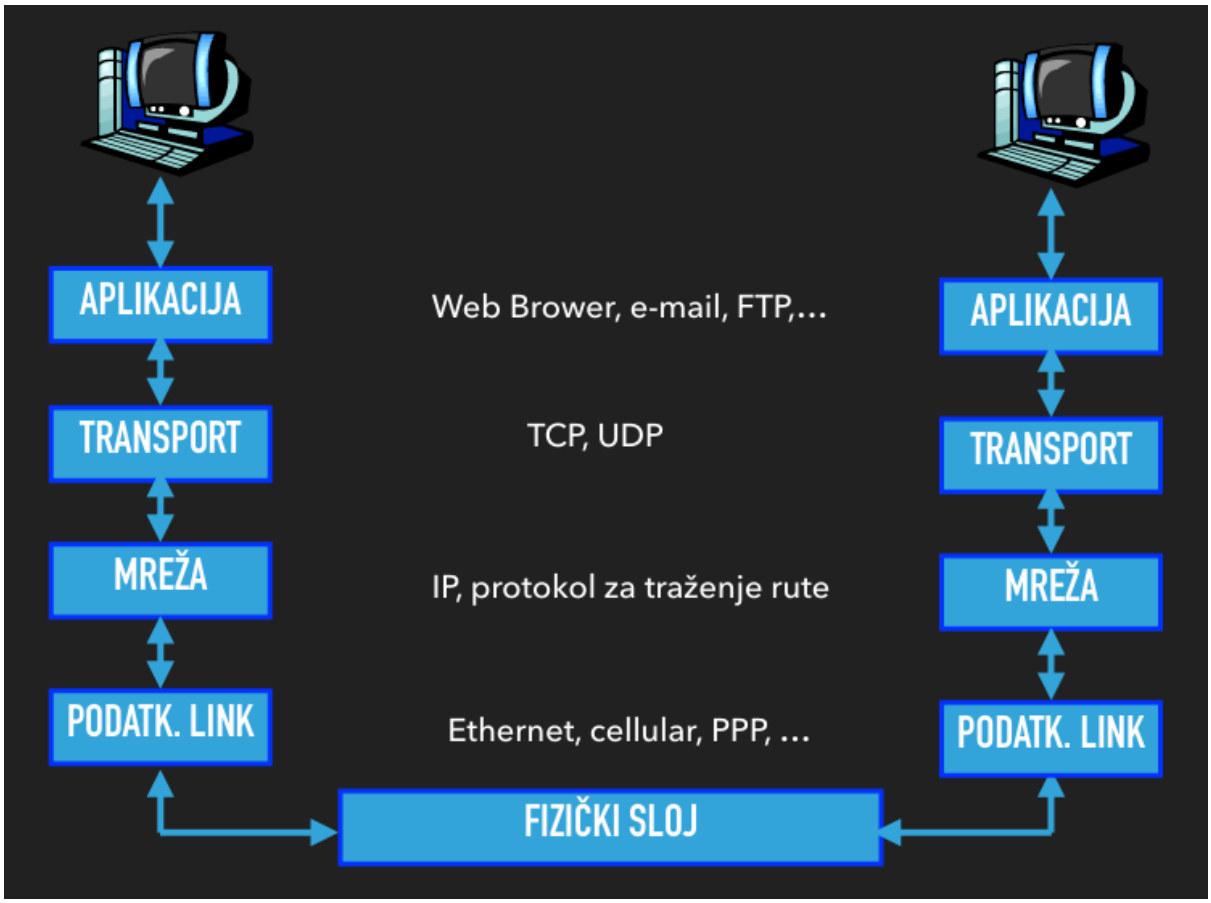
```
        std::unique_lock<std::mutex> l(lock);
        not_empty.wait(l, [this](){return count != 0; });
        int result = buffer[front];
        front = (front + 1) % capacity;
        --count;
        l.unlock();
        not_full.notify_one();
        return result;
    }
};
```

```
void consumer(int id, BoundedBuffer& buffer) {
    for(int i = 0; i < 50; ++i){
        int value = buffer.get();
        std::cout << "Consumer " << id << " took " << value << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
    }
}

void producer(int id, BoundedBuffer& buffer){
    for(int i = 0; i < 75; ++i){
        buffer.put(i);
        std::cout << "Producer " << id << " produced " << i << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

int main(){
    BoundedBuffer buffer(200);

    std::thread c1(consumer, 0, std::ref(buffer));
    std::thread c2(consumer, 1, std::ref(buffer));
    std::thread c3(consumer, 2, std::ref(buffer));
    std::thread p1(producer, 0, std::ref(buffer));
    std::thread p2(producer, 1, std::ref(buffer));

    c1.join(); c2.join(); c3.join();
    p1.join(); p2.join();

    return 0;
}
```

The `BoundedBuffer` class completely encapsulates the synchronization primitives and enables simple use by using the `put` and `get` methods.

11 Introduction to Internet Communications

11.1 Exchange of "Packets" on the Internet



Slika 127: A simplified model of internet packet exchange.

11.2 Transport Layer

- **TCP (analogous to telephone communication)**
  - **Connection-oriented:** A connection must be established before sending data.
  - After establishment, data is written or read with simple `write/read` methods.
  - **Stream-oriented:** The application simply writes data; TCP buffers the data and divides it into packets that it sends/receives.
  - **Reliable:** Input data is broken down into packets (max 1500 bytes), TCP numbers each packet and delivers them in the correct order. Guaranteed correctness and order of data.
  - Web, e-mail, client-server paradigm.
- **UDP (analogous to postal service)**
  - The user divides the message into packets themselves (max 1500 bytes).
  - Each packet requires correct addressing.
  - Delivery and packet order are not guaranteed.

- Since there is no acknowledgment of receipt, communication is faster.
- Suitable for RT applications that allow the loss of a small part of the information (multimedia, VoIP,...).

11.3 Socket vs. File Descriptor

- For network communication, it is necessary to create a socket, which can be seen as a FILE descriptor in the C programming language.
- After creating a socket, data is written to it or read from it, which has been delivered via the network interface.
- For example, creating a socket for TCP communication is:

```
int fd; // socket id
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    // Error! Cannot create socket
    return;
}

```

- The `socket` method returns an integer that also serves to check for correctness:
  - `fd < 0` error, incorrect identifier, socket cannot be created.
  - `fd >= 0` a correct identifier has been assigned.
- Other definitions:
  - **AF_INET:** The Internet family protocol is assigned to the socket.
  - **SOCK_STREAM:** The socket is set for the TCP protocol.
  - **SOCK_DGRAM:** The socket is set for the UDP protocol.
- Creating a socket for UDP communication is:

```
int fd; // socket id
if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    // Error! Cannot create socket
    return;
}

```

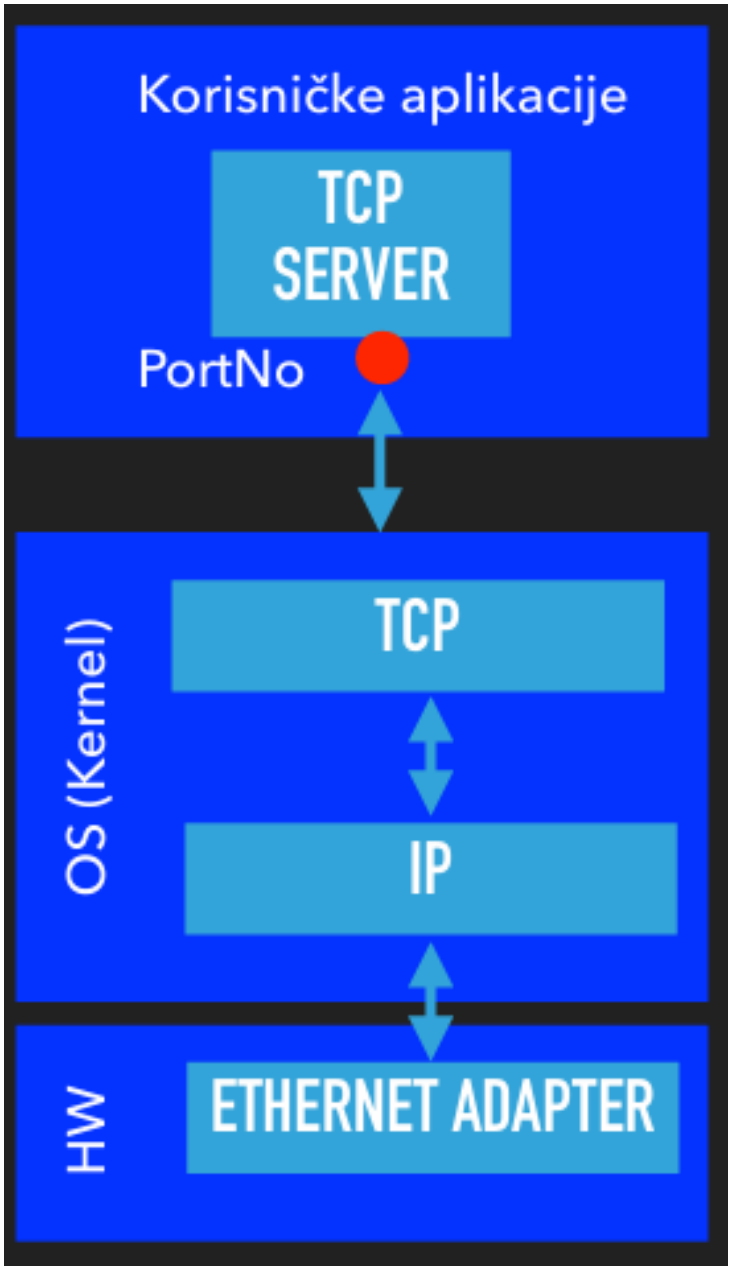
11.4 Client-Server: TCP Server

The client-server paradigm requires:

- One server application that runs on a machine on the network and which works without interruption, serving client calls.
- One or more client applications that send requests to the server.
  - The client initiates communication.

The implementation of a server application can be explained by an analogy to a manual telephone exchange in an organization:

1. It is necessary to install a switchboard with a telephone number (create a `socket`).
2. Assign a phone number to the incoming call (`bind`).
3. Listen for incoming calls (`listen`).
4. After accepting a call, connect it to the requested extension (`accept`).
5. The requested user listens for the request (`read`) and responds to it (`write`).
6. If the command to end the call has not been received, return to position 3 and wait for a new connection.



Slika 128: TCP Server architecture.

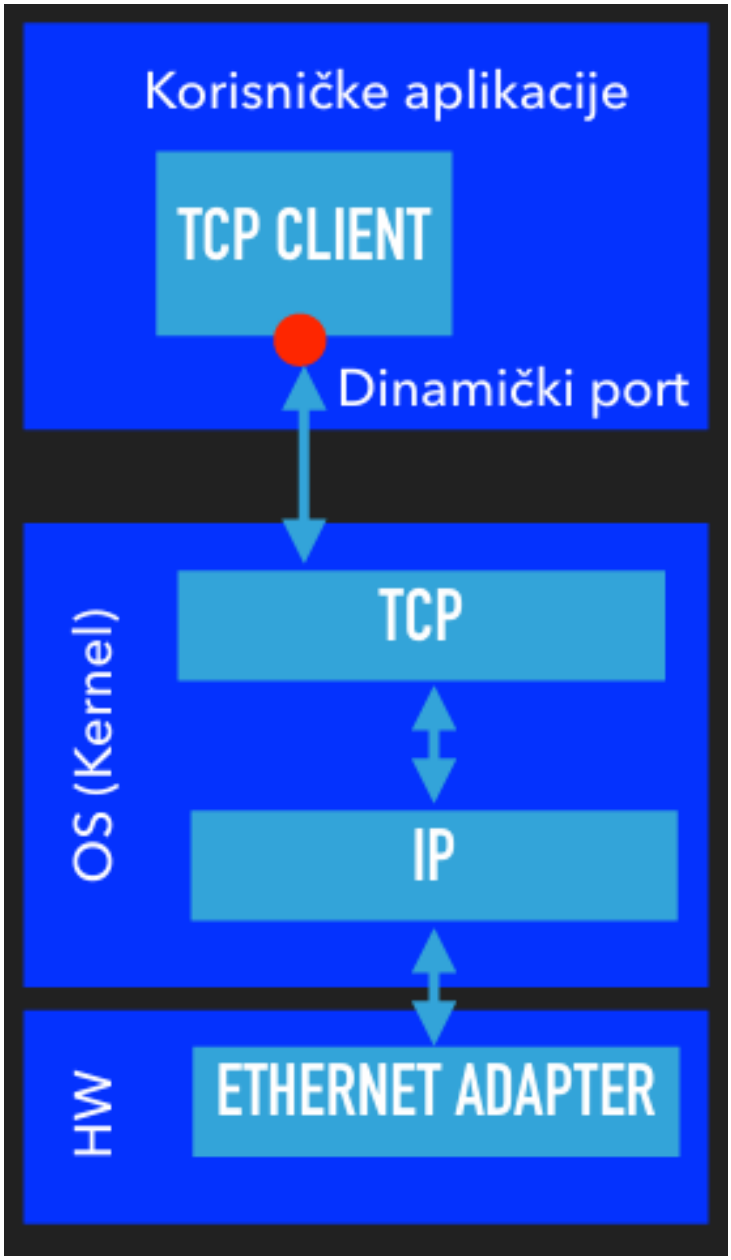
11.4.1 TCP Server - C Implementation

```
int fd; // socket id
struct sockaddr_in srv; // Address and port of the server "listening"
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return; // Error! Cannot create socket

srv.sin_family = AF_INET; // Internet family
srv.sin_port = htons(8085); // server will listen on port 8085
// bind: client can connect to any of my addresses
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0)
    return; // Error! Cannot bind

// listening for incoming calls
// defines how many incoming calls are kept "on hold"
if(listen(fd, 5) < 0)
    return;

struct sockaddr_in cli; // client information
int clientFD;           // socket for new connection
int clientLen = sizeof(cli); // length of cli structure

while (true) {
    clientFD = accept(fd, (struct sockaddr*) &cli, &clientLen);
    if(clientFD < 0)
        return;

    char buf[512]; // buffer for data to be read
    int nBytes;    // number of bytes read

    // read blocks the application until data arrives
    if((nBytes = read(clientFD, buf, sizeof(buf))) < 0)
        printf("Error");
}
```

11.5 Client-Server: TCP Client

The implementation of a client application can be explained by an analogy to a telephone call:

1. A device with a phone number and the phone number of the person being called is required (create a `socket` and assign the server's internet address and port).
2. Call the user's number (`connect`).
3. Conversation (`write`).
4. Conversation (`read`).
5. ....
6. End the connection (`closesocket`).



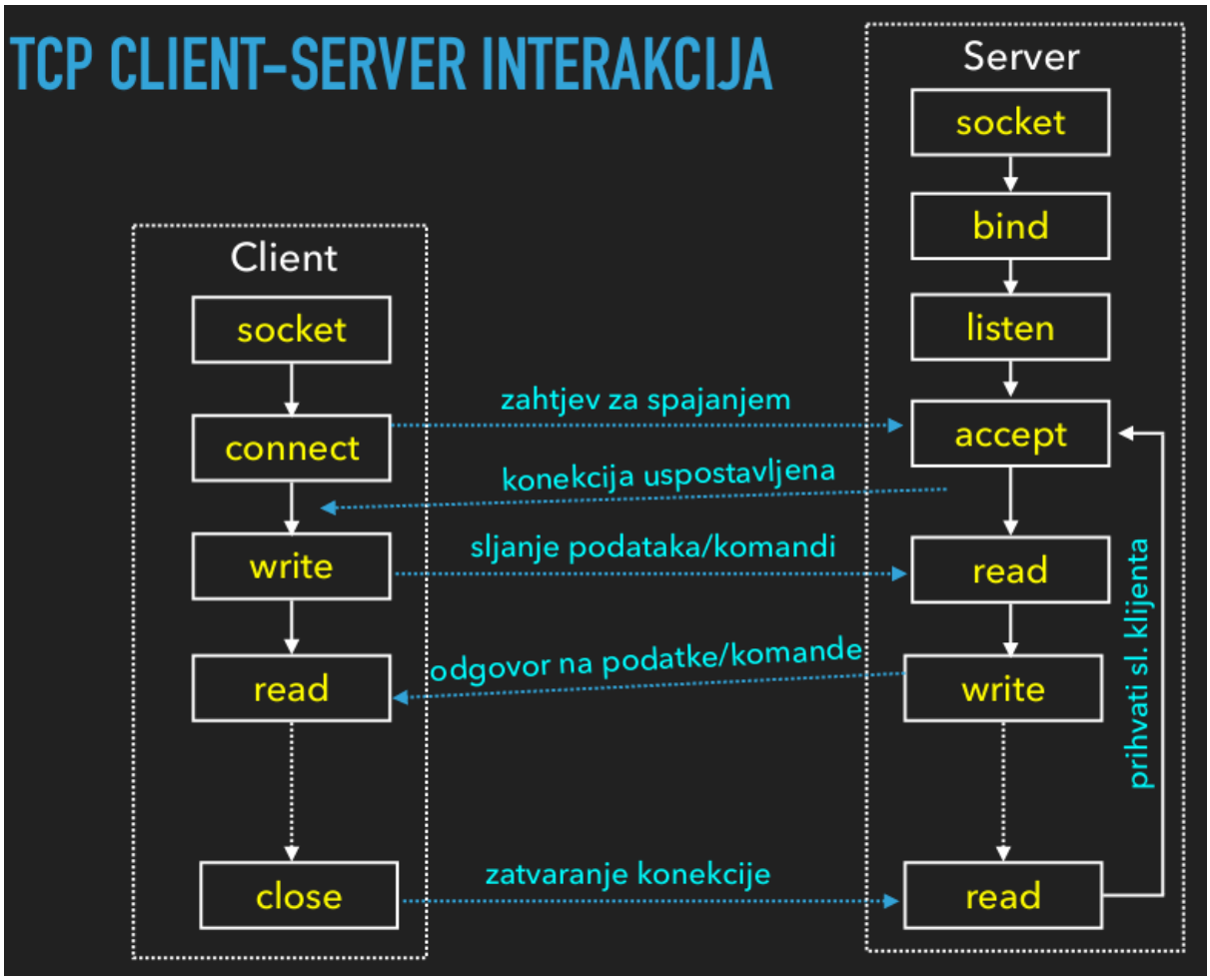
Slika 129: TCP Client architecture.

11.5.1 TCP Client - C Implementation

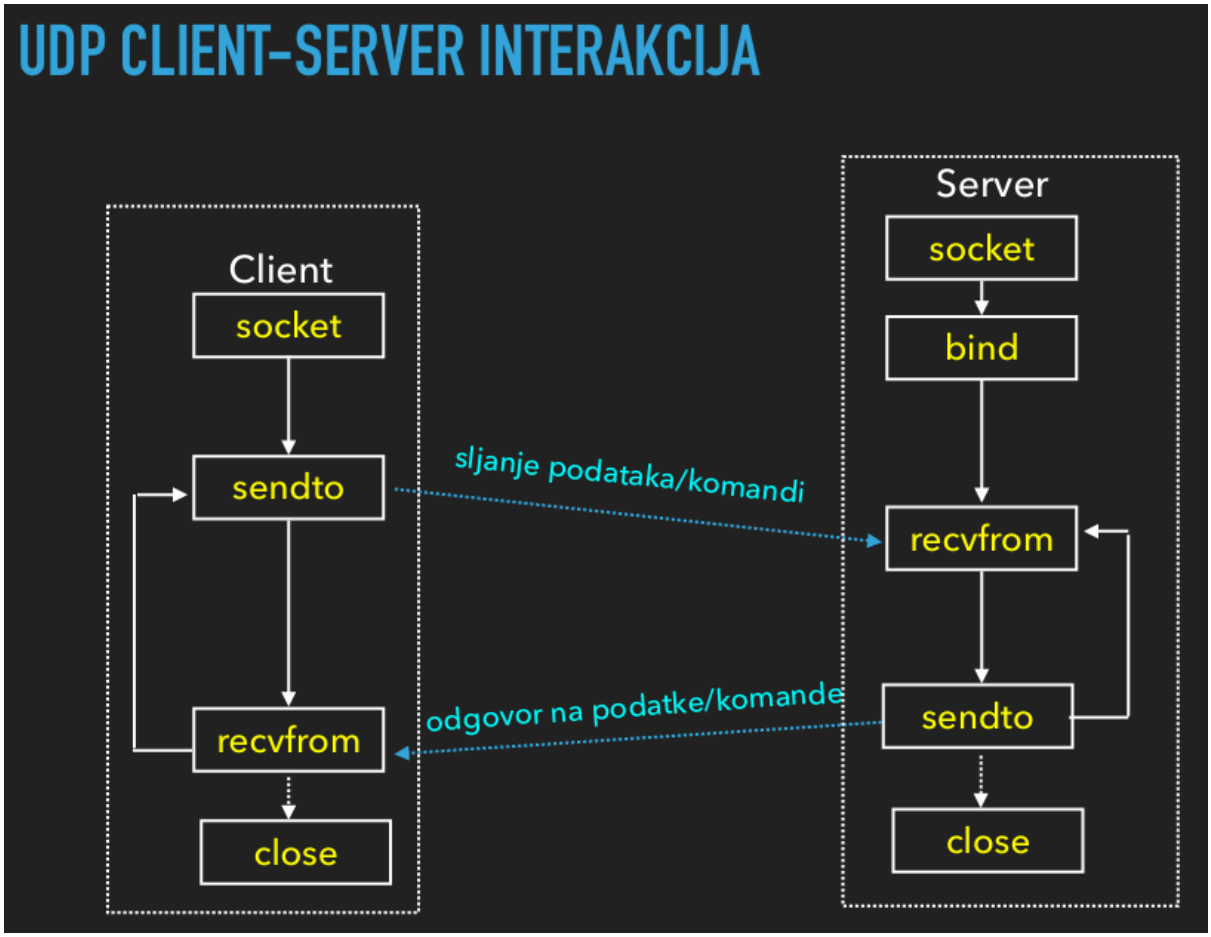
```
int fd; // socket descriptor
struct sockaddr_in srv; // address of the server to connect to

// 1. create socket
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return;

// 1.1 prepare address family
srv.sin_family = AF_INET;
// and the port on which the server is registered
srv.sin_port = htons(8085);
// convert string "192.168.0.50" to server address
srv.sin_addr.s_addr = inet_addr("192.168.0.50");

```

```
// 2. connect
if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0)
    return;

char buf[512]; // buffer for data being sent
int nBytes = 0; // number of bytes being sent
// initialize nBytes = len(header) + len(data)
int nWritten = 0;

if((nWritten = write(fd, buf, nBytes)) < 0)
    return;

// ... read response from server
```

11.6 TCP and UDP Interaction Summary



Slika 130: TCP Client-Server Interaction Flowchart.



Slika 131: UDP Client-Server Interaction Flowchart.

11.7 UDP Server and Client - C Implementation

```
int socketID; // socket descriptor
if((socketID = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    return;

struct sockaddr_in srv; // needed for bind
// using Internet address/protocol family
srv.sin_family = AF_INET;
// set the port to listen on
srv.sin_port = htons(80);
// set the server so that the client can connect
// to any of my addresses
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(socketID, (struct sockaddr*)&srv, sizeof(srv)) < 0)
    return;

struct sockaddr_in cli; // used by recvfrom()
char buf[512];          // used by recvfrom()
int cli_len = sizeof(cli);
int nbytes;

while (true)
{
    if (nbytes = recvfrom(socketID, buf, sizeof(buf), 0,
```

```
                        (struct sockaddr*) &cli, &cli_len))
        return;
    // ... process data stored in buf
}
```

Listing 1: UDP Server - C Implementation

```
int socketID; // socket descriptor
if((socketID = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    return;

struct sockaddr_in srv; // used by sendto()
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("192.168.0.50");

// sendto: send data to IP address "192.168.0.50" port 80
nbytes = sendto(socketID, buf, sizeof(buf), 0,
                (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0)
{
    return;
}
```

Listing 2: UDP Client - C Implementation

11.8 Cyber Security: Buffer Overflow Attack (BOA)

- BOA is considered the most commonly used method of attack from remote machines.
- The goal of the attack is to change the execution flow of the application, i.e., to call injected malicious binary code.
- They can be stack and heap overflow-based attacks.
- As with many attacks, the problem lies in the fact that software vulnerabilities are exploited by mixing data and control information, thereby changing the program's execution flow.
- The attack can be carried out locally or over a network (TCP/IP, UDP) and is based on sending long input to an application where control information is written.
- An example of a cyber attack will be explained on a stack-based buffer overflow example.

```
ElemType arr[100]; // array of 100 elements allocated on the stack
for (int i = 0; i < 120; ++i)
{
    // Buffer overflow! 120 elements are written to a buffer of max 100
    arr[i] = ...
}

```

- BOA exploits programmer carelessness and a fact related to the C/C++ calling convention.
  - When a function is called, its parameters are placed on the stack with local variables.
  - After that, the return address is written so that the CPU knows where the procedure was called from and can continue from that position.

  - If there is a local buffer within the procedure, into which a programmer's carelessness allows writing outside its bounds, then instead of the return address, another address can be written that calls the injected code, and thus a successful buffer overflow attack is achieved.

- Typical C functions where mistakes are made are:
  - `memcpy, strcpy, gets, strcat, sprintf, printf, fprintf,...`
- In addition to this, it must be ensured that direct access to the buffer does not write more than the maximum length of the buffer.
  - Using an array class with operator `[]` overloading instead of a C array.

Slika 132: If a write to Arr is performed beyond its bounds (N), it is possible to change the content of the field that contains the function's return address.