ДОДАТОК А «ЛІСТИНГ ПРОГРАМИ»

```
#ifndef ECHO_INTERFACES_H
#define ECHO_INTERFACES_H

#include <vector>
#include <condition_variable>
#include <mutex>
#include <thread>
#include <map>
#include <fstream>

enum Result
{
    ok,
};

/**
 * @brief Class for launching voice listener
 */
class VoiceListener
{
public:
    VoiceListener(std::mutex &mutex_,
                  std::condition_variable &conditionalVariable,
                  std::vector<std::string> &recognizedWords_);

    virtual  ~VoiceListener() = default;

public:

    virtual Result start() = 0;

protected:
    std::mutex & mutex_;
    std::condition_variable & conditionVariable_;
    std::vector<std::string> & recognizedWords_;
};

/**
 * @brief Class for launching CMUSphinx in terminal and setup listener.
 */
class CMUSphinxVoiceListener : public VoiceListener
{
public:
    CMUSphinxVoiceListener(std::mutex &mutex_,
                           std::condition_variable &conditionalVariable,
                           std::vector<std::string> &recognizedWords_);

    Result start() override;
};

/**
 * @brief Class for validation commands
 */
class CommandValidator
{
public:
```

```
    CommandValidator(std::mutex &mutex_,
                     std::condition_variable &conditionalVariable,
                     std::vector<std::string> &recognizedWords_);

    virtual ~CommandValidator() = default;

public:

    virtual Result initializeSupportedCommandNames(const std::string &
pathToToolNamesPreset) = 0;
    virtual Result validateRecognizedCommand() = 0;

private:
    std::mutex & mutex_;
    std::condition_variable & conditionVariable_;
    std::vector<std::string> & recognizedWords_;

    std::vector<std::string> supportedToolNames_;
};

/**
 * @brief Class for validation tool commands
 */
class ToolCommandValidator : public CommandValidator
{
public:
    ToolCommandValidator(std::mutex &mutex_,
                     std::condition_variable &conditionalVariable,
                     std::vector<std::string> &recognizedWords_);

public:
    Result initializeSupportedCommandNames(const std::string &
pathToToolNamesPreset) override;
    Result validateRecognizedCommand() override;
};

class NmapToolCommandValidator : public ToolCommandValidator
{
public:
    NetstatToolCommandValidator(std::mutex &mutex_,
    std::condition_variable &conditionalVariable,
            std::vector<std::string> &recognizedWords_);

public:
    Result initializeSupportedCommandNames(const std::string &
pathToToolNamesPreset) override;
    Result validateRecognizedCommand() override;
};

class NetstatToolCommandValidator : public ToolCommandValidator
{
public:
    NetstatToolCommandValidator(std::mutex &mutex_,
    std::condition_variable &conditionalVariable,
            std::vector<std::string> &recognizedWords_);

public:
    Result initializeSupportedCommandNames(const std::string &
pathToToolNamesPreset) override;
    Result validateRecognizedCommand() override;
};

using ApplicationParameterName = std::string;
using ApplicationParameterDescription = std::string;
```

```
class Application
{
public:
    explicit Application(const std::string & applicationName);
    ~Application() = default;

public:
    Result  start() const;
    void    stop()  const;

public:
    Result initializeRequiredParameters(std::map<ApplicationParameterName,
ApplicationParameterDescription> & requiredParameters);

    void setOutputFilename(const std::string & filename);
    void setParameters(const std::string & parameters);

    std::string getApplicationName() const;
    std::string getOutputFilename() const;
    std::string getProvidedParameters() const;
    uint32_t getApplicationPID() const;
private:

    std::string applicationName_;
    std::string outputFileName_;

    std::map<ApplicationParameterName, ApplicationParameterDescription>
requiredParameters_;
    std::string providedParameters_;
    std::ifstream outputFile_;

    uint32_t currentPID_;
};

class Window
{
public:
    explicit Window(Ui::MainWindow * ui);
    ~Window() = default;

    [[nodiscard]] Ui::MainWindow * getUi() const;

protected:
    Ui::MainWindow * ui_;
};

class StatusWindow : public Window
{
public:
    enum class WindowsStatus
    {
        SUCCESS = 0,  ///< Represents the action, that was successfully done,
and we want to inform user about that
        WARNING,      ///< Represents the action, that was user has to be
warned about
        ERROR,        ///< Represents the action, that was failed, and we
want to inform user about that
        UNDEFINED
    };
public:
    Result setStatusExplanation(const std::string & statusExplanation);
    Result setStatus(WindowsStatus currentWindowStatus);
};
```

```
class DataRequestWindow : public Window
{
public:
    Result addRadioButtonsRow(const std::string & rowDescription,
                              const std::vector<std::string> &
radioButtonsRow);

    Result addInputField(const std::string & inputDescription);
};

/**
 * Class for generating interactive windows for user
 */
class WindowGenerator
{
protected:
    WindowGenerator() = default;

    static std::unique_ptr<WindowGenerator> windowsGenerator_;

public:
    static WindowGenerator* getInstance();

public:
    virtual Result generateWindow(const Window & window) = 0;

};

/**
 * Class for generation Error/Warning/Success popups for user
 */
class StatusWindowGenerator : public WindowGenerator
{
public:
    Result generateWindow(const Window & window) override;

public:
    Result setStatusExplanation(StatusWindow & window, const std::string &
statusExplanation);
    Result setStatus(StatusWindow & window, StatusWindow::WindowsStatus
currentWindowStatus);

};

/**
 * Class for generation popup with question and input boxes(buttons, input
fields). Or sequences of such popups
 */
class DataRequestWindowGenerator : public WindowGenerator
{
public:
    Result generateWindow(const Window & window) override;

    static Result addRadioButtonsRow(DataRequestWindow & window,
                              const std::string & rowDescription,
                              const std::vector<std::string> &
radioButtonsRow);

    static Result addInputField(DataRequestWindow & window, const std::string
& inputDescription);

};
```

```
/**
 * @brief Class for logging information such as ERRORS/WARNINGS/INFO messages
into a log file
 */
class Logging
{
public:
    enum LoggingLevel
    {
        LOG_DISABLED,   ///< Disable all traces
        LOG_ERROR,      ///< Errors, enabled by default.
        LOG_WARNING,    ///< Warnings, enabled by default.
        LOG_INFO,       ///< Important information, enabled by default.
        LOG_DEBUG,      ///< Verbose information (high amount of information,
debugging only).
        NUM_LOGS        ///< It tells us how many log levels have been
defined
    };
public:

    Logging () = default;
    virtual ~Logging () = default;

    /**
     * @brief Returns the currently enabled logging level
     * @return The current logging level threshold
     */
    [[nodiscard]] LoggingLevel getLevel() const;

    void setLevel (LoggingLevel level);

    /**
     * See LoggingBD.h
     */
    void logDebug   (const char * message, ...) const;
    void logInfo    (const char * message, ...) const;
    void logWarning (const char * message, ...) const;
    void logError   (const char * message, ...) const;

};

/**
 * @brief Class for managing tool output parameters
 */
class ToolOutputManager
{
public:
    ToolOutputManager() = default;

    ~ToolOutputManager() = default;

public:

private:
    StatusWindowGenerator dataRequestWindowGenerator_;
};

/**
 * @brief Class for managing tool input parameters.
 */
class ToolInputManager
{
public:
    ToolInputManager() = default;
```

```
    ~ToolInputManager() = default;


public:
    Result requestApplicationDataFromUser(Application & application);

private:
    DataRequestWindowGenerator dataRequestWindowGenerator_;

};

using ApplicationName = std::string;

/**
 * @brief Class for managing tools lifecycle(launch, kill)
 */
class ToolManager
{
public:
    ToolManager() = default;
    ~ToolManager() = default;
public:
    Result startApplication(std::string & applicationName);
    Result stopApplication (Application & application);
    Result stopApplication (uint32_t applicationPID);

    Result initializeSupportedApplicationNames(const std::string &
pathToApplicationNamesPreset);

    Result requestApplicationParametersFromUser(Application & application);

    void listAllRunningApplications() const;
private:
    ToolOutputManager toolOutputManager_;
    ToolInputManager  toolInputManager_;

    std::map<ApplicationName, Application> runningApplications_;
    std::vector<std::string> supportedApplicationNames_;
};


#endif //ECHO_INTERFACES_H
```