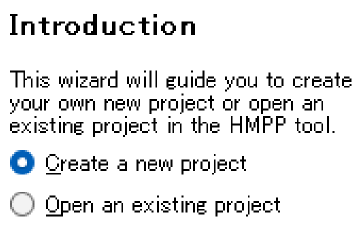
CRCI Tool User Manual

This user manual provides instructions for using the CRCI tool, which is part of the entire tool to support both programming and program review. Since the entire tool includes multiple features, this manual focuses exclusively on CRCI-specific functionality; other features are not covered here.

Create Project

Upon launching the tool, users will be prompted to create or import a project (see Figure 1).



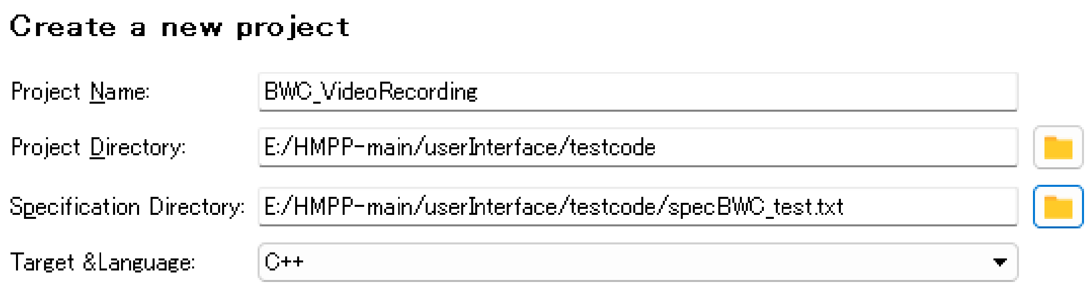
Introduction

This wizard will guide you to create your own new project or open an existing project in the HMPP tool.

[✓ Create a new project]

Open an existing project

Figure 1



Create a new project

Project Name:	BWC_VideoRecording
Project Directory:	E:/HMPP-main/userInterface/testcode
Specification Directory:	E:/HMPP-main/userInterface/testcode/specBWC_test.txt
Target &Language:	C++

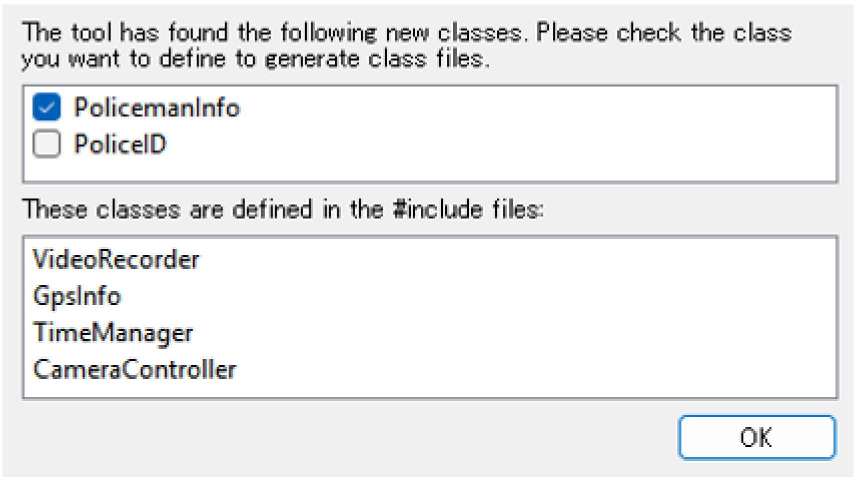
Figure 2

If users choose to import a project, it must have been created previously in the tool, and users will need to select its *CMakeLists.txt* file. If no specification is available, users do not need to provide one.

CRCI-Tool

After creating or importing a project, users can use the menu bar to open or create files for the current project, write code, and perform other standard operations, similar to those in a typical code editor. These basic operations are not described in detail in this manual. The following sections focus on the CRCI tool features.

When users have finished editing the current file and wish to run the CRCI tool, first save the file (press Ctrl+S or click the Save button in the menu bar). A window will appear (Figure 3) that includes additional features; the user may simply click OK to proceed. The tool will then analyze the functions defined in the current file along with their corresponding code.


The tool has found the following new classes. Please check the class you want to define to generate class files.

- [✓] PolicemanInfo
- [ ] PoliceID

These classes are defined in the #include files:

VideoRecorder
GpsInfo
TimeManager
CameraController

Figure 3

Next, click the CRCI tool button (Figure 4).

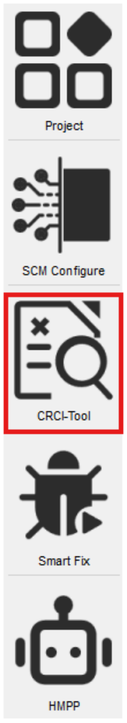
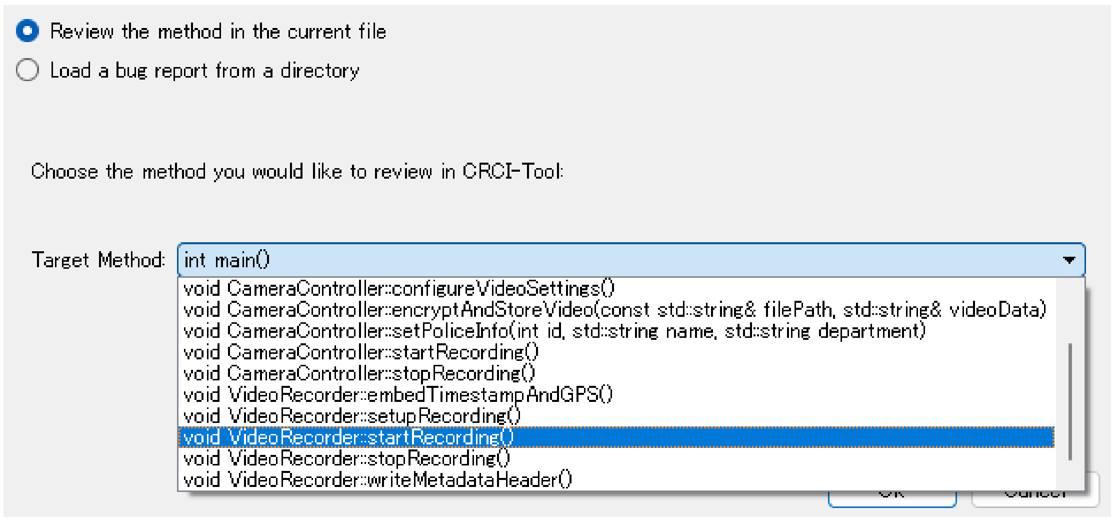


Figure 4

A window will appear, allowing the user to choose between:

- Reviewing code
- Importing and viewing a review report



[✓ Review the method in the current file]

Load a bug report from a directory

Choose the method you would like to review in CRCI-Tool:

Target Method: int main()

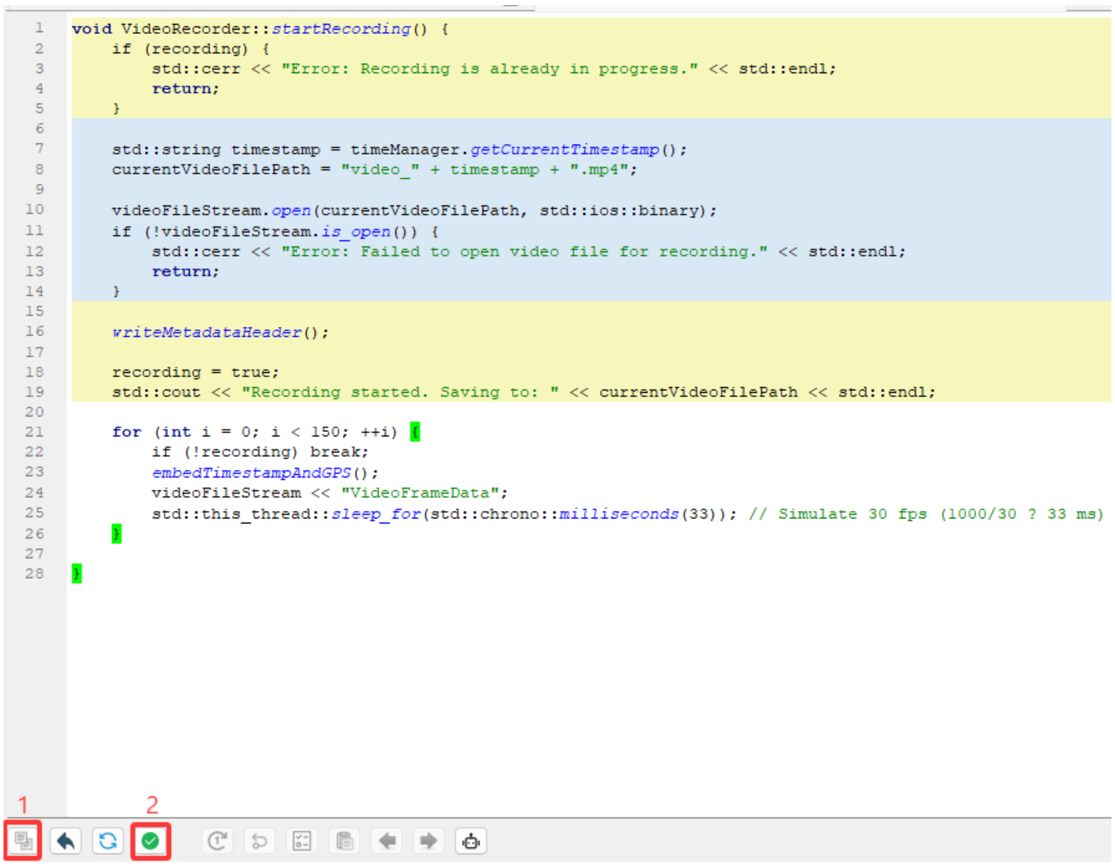
```
void CameraController::configureVideoSettings()
void CameraController::encryptAndStoreVideo(const std::string& filePath, std::string& videoData)
void CameraController::setPoliceInfo(int id, std::string name, std::string department)
void CameraController::startRecording()
void CameraController::stopRecording()
void VideoRecorder::embedTimestampAndGPS()
void VideoRecorder::setupRecording()
void VideoRecorder::startRecording()
void VideoRecorder::stopRecording()
void VideoRecorder::writeMetadataHeader()
```

Figure 5

Review Code

If the user chooses to review code, they may review all code in the current file or only a specific function.

The user can then segment the code by function:

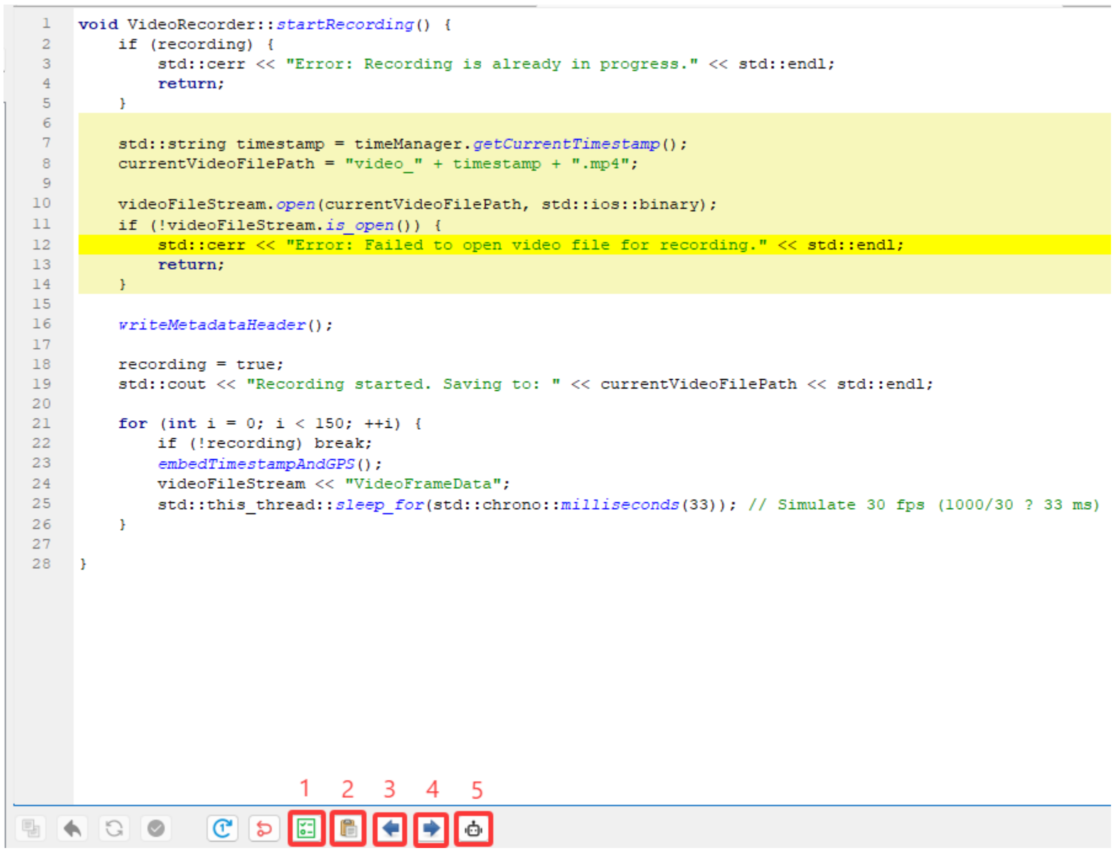


```cpp
void VideoRecorder::startRecording() {
    if (recording) {
        std::cerr << "Error: Recording is already in progress." << std::endl;
        return;
    }

    std::string timestamp = timeManager.getCurrentTimestamp();
    currentVideoFilePath = "video_" + timestamp + ".mp4";

    videoFileStream.open(currentVideoFilePath, std::ios::binary);
    if (!videoFileStream.is_open()) {
        std::cerr << "Error: Failed to open video file for recording." << std::endl;
        return;
    }

    writeMetadataHeader();

    recording = true;
    std::cout << "Recording started. Saving to: " << currentVideoFilePath << std::endl;

    for (int i = 0; i < 150; ++i) {
        if (!recording) break;
        embedTimestampAndGPS();
        videoFileStream << "VideoFrameData";
        std::this_thread::sleep_for(std::chrono::milliseconds(33)); // Simulate 30 fps (1000/30 ? 33 ms)
    }

}
```

Figure 6

Click Button 1 (Figure 6) to divide the code into segments. Each segment will be highlighted in a

different color.
Click Button 2 (Figure 6) to complete code segmentation and enter the review phase.

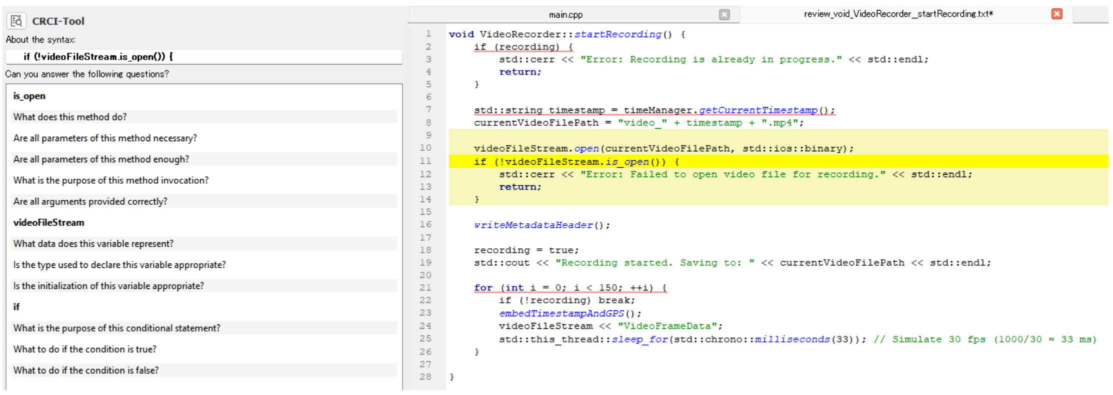


```cpp
void VideoRecorder::startRecording() {
    if (recording) {
        std::cerr << "Error: Recording is already in progress." << std::endl;
        return;
    }

    std::string timestamp = timeManager.getCurrentTimestamp();
    currentVideoFilePath = "video_" + timestamp + ".mp4";

    videoFileStream.open(currentVideoFilePath, std::ios::binary);
    if (!videoFileStream.is_open()) {
        std::cerr << "Error: Failed to open video file for recording." << std::endl;
        return;
    }

    writeMetadataHeader();

    recording = true;
    std::cout << "Recording started. Saving to: " << currentVideoFilePath << std::endl;

    for (int i = 0; i < 150; ++i) {
        if (!recording) break;
        embedTimestampAndGPS();
        videoFileStream << "VideoFrameData";
        std::this_thread::sleep_for(std::chrono::milliseconds(33)); // Simulate 30 fps (1000/30 ? 33 ms)
    }

}
```

Figure 7

Next, the tool guides the user through the code review process. By default, the review proceeds line by line, starting from the first code segment. Of course, users may also choose to begin the review from any code segment. In such cases, the tool assumes that all preceding segments have already been reviewed. The current segment is highlighted in dark yellow, while the specific line under review is highlighted in light yellow to help the user focus on the relevant code, as illustrated in Figure 7.

Once a segment has been fully reviewed, the user can click Button 1 (Figure 7) to automatically jump to the first line of the next segment. After all segments have been reviewed, the tool prompts the user to generate a review report or the user can directly generate a review report by clicking Button 2 (Figure 7).



CRCI-Tool

About the syntax:

if (!videoFileStream.is_open()) {

Can you answer the following questions?

is_open

What does this method do?

Are all parameters of this method necessary?

Are all parameters of this method enough?

What is the purpose of this method invocation?

Are all arguments provided correctly?

videoFileStream

What data does this variable represent?

Is the type used to declare this variable appropriate?

Is the initialization of this variable appropriate?

if

What is the purpose of this conditional statement?

What to do if the condition is true?

What to do if the condition is false?

main.cpp | review_void_VideoRecorder_startRecording.txt*

```cpp
void VideoRecorder::startRecording() {
    if (recording) {
        std::cerr << "Error: Recording is already in progress." << std::endl;
        return;
    }

    std::string timestamp = timeManager.getCurrentTimestamp();
    currentVideoFilePath = "video_" + timestamp + ".mp4";

    videoFileStream.open(currentVideoFilePath, std::ios::binary);
    if (!videoFileStream.is_open()) {
        std::cerr << "Error: Failed to open video file for recording." << std::endl;
        return;
    }

    writeMetadataHeader();

    recording = true;
    std::cout << "Recording started. Saving to: " << currentVideoFilePath << std::endl;

    for (int i = 0; i < 150; ++i) {
        if (!recording) break;
        embedTimestampAndGPS();
        videoFileStream << "VideoFrameData";
        std::this_thread::sleep_for(std::chrono::milliseconds(33)); // Simulate 30 fps (1000/30 ≈ 33 ms)
    }

}
```

Figure 8

While the user is reviewing a line of code, the tool presents a set of questions intended to prompt reflection on potential issues as shown in Figure 8. By answering these questions, developers are guided to identify possible bugs in their code.

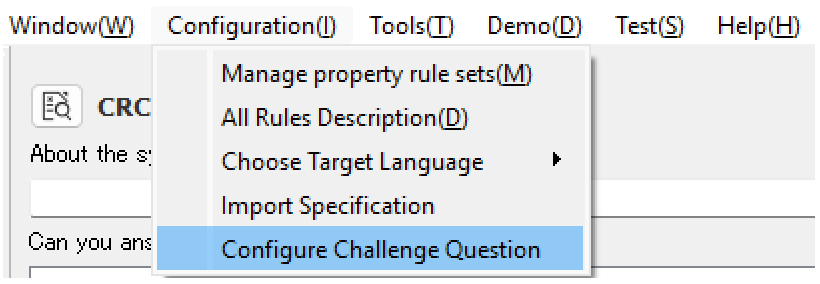
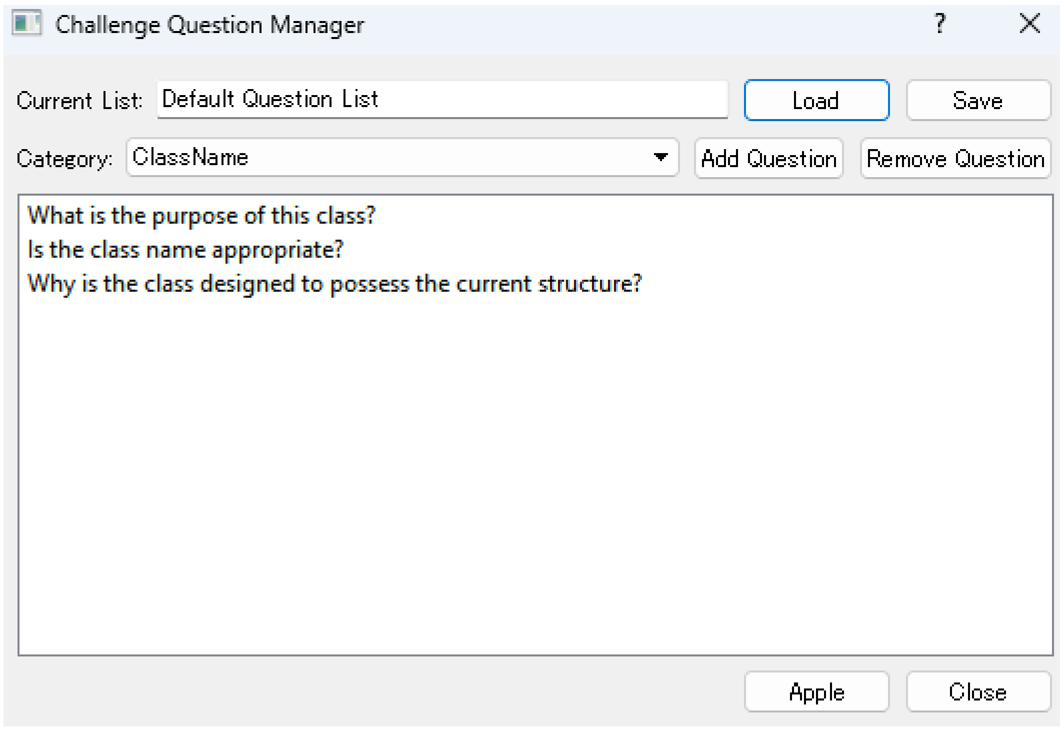


Figure 9

For these pre-prepared questions, users can also create their own versions. Click Configure Challenge Question in the menu bar (Figure 9). A window will appear (Figure 10), allowing users to import or export the question list for the current project, as well as add or delete questions in each category.

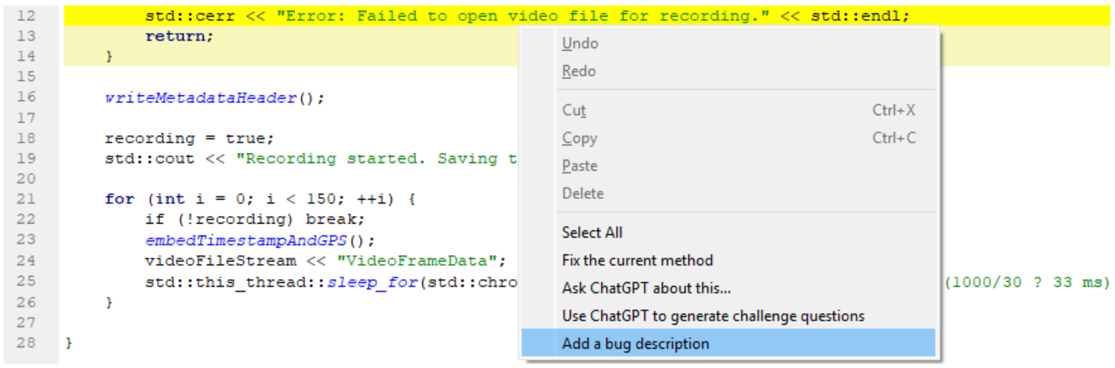


Current List: Default Question List

Category: ClassName

What is the purpose of this class?
Is the class name appropriate?
Why is the class designed to possess the current structure?

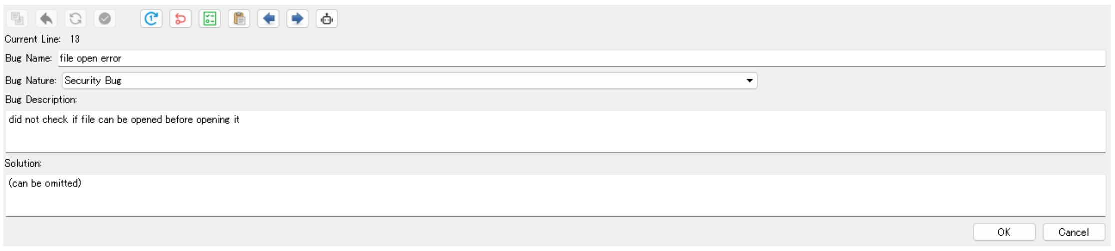
Figure 10



```
        std::cerr << "Error: Failed to open video file for recording." << std::endl;
        return;
    }

    writeMetadataHeader();

    recording = true;
    std::cout << "Recording started. Saving t

    for (int i = 0; i < 150; ++i) {
        if (!recording) break;
        embedTimestampAndGPS();
        videoFileStream << "VideoFrameData";
        std::this_thread::sleep_for(std::chro                                    (1000/30 ? 33 ms)
    }
}
```

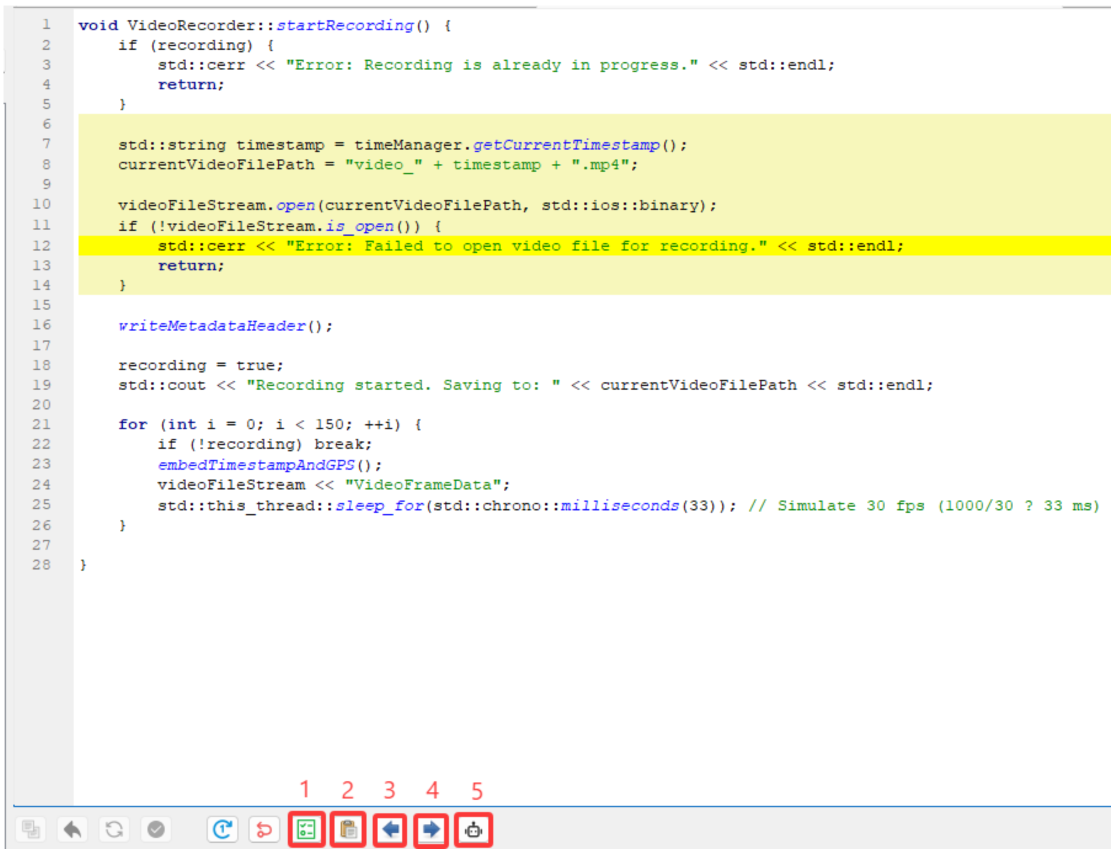
Figure 11

If the user discovers a bug in the current row, they can right-click to open a pop-up window (Figure 11). Select Add a Bug Description, then enter the relevant details about the bug (Figure 12). Click OK to submit the bug description.



Current Line: 13
Bug Name: file open error
Bug Nature: Security Bug
Bug Description:
did not check if file can be opened before opening it
Solution:
(can be omitted)

Figure 12



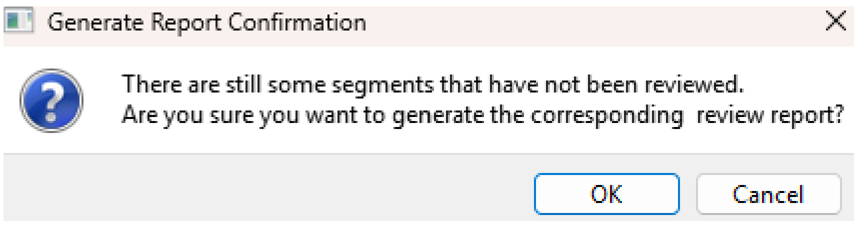
```cpp
void VideoRecorder::startRecording() {
    if (recording) {
        std::cerr << "Error: Recording is already in progress." << std::endl;
        return;
    }

    std::string timestamp = timeManager.getCurrentTimestamp();
    currentVideoFilePath = "video_" + timestamp + ".mp4";

    videoFileStream.open(currentVideoFilePath, std::ios::binary);
    if (!videoFileStream.is_open()) {
        std::cerr << "Error: Failed to open video file for recording." << std::endl;
        return;
    }

    writeMetadataHeader();

    recording = true;
    std::cout << "Recording started. Saving to: " << currentVideoFilePath << std::endl;

    for (int i = 0; i < 150; ++i) {
        if (!recording) break;
        embedTimestampAndGPS();
        videoFileStream << "VideoFrameData";
        std::this_thread::sleep_for(std::chrono::milliseconds(33)); // Simulate 30 fps (1000/30 ? 33 ms)
    }

}
```

Figure 7

Users can click Buttons 3 and 4 (Figure 7) to navigate through the submitted bug information, and the tool will automatically jump to the corresponding line in the code.

Users can also click Button 5 (Figure 7) to call ChatGPT to detect bugs in the current line and automatically fill the results into the bug information. However, this requires prior configuration of the *OpenAIkey.json* file in the tool's *config* directory to ensure the key is available.

When generating a review report, if any code segments remain unreviewed, the tool will prompt the user to confirm whether to proceed (Figure 13). The user can then select the directory in which to save the report.



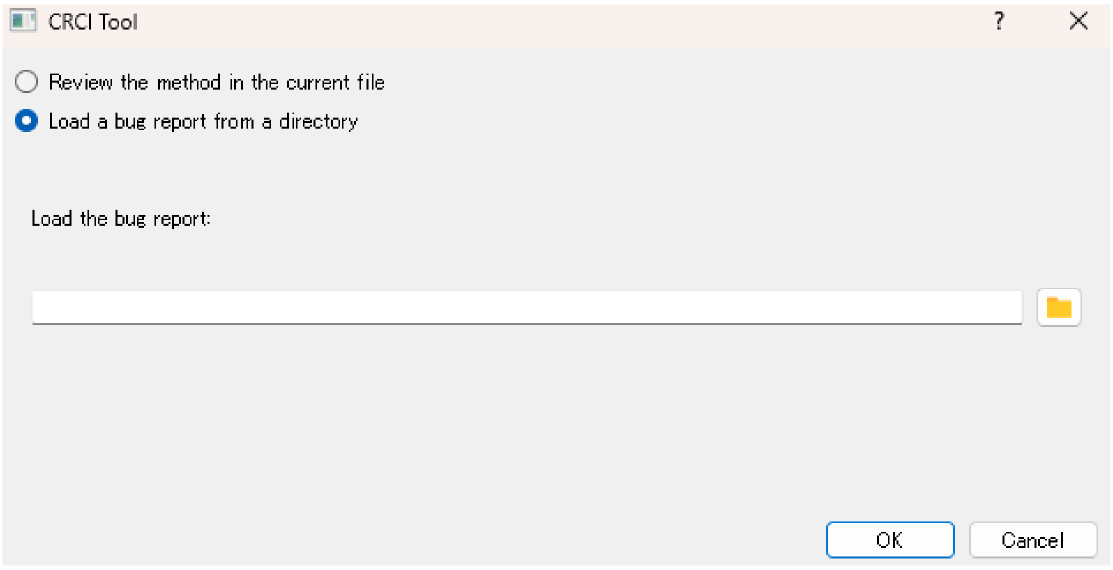
Generate Report Confirmation

There are still some segments that have not been reviewed.
Are you sure you want to generate the corresponding review report?

Figure 13

This report compiles all relevant information, including the reviewed code and associated bug data and saves it in a file named *review_methodName.txt* in JSON format.

View Review Report


CRCI Tool

( ) Review the method in the current file
[✓ Load a bug report from a directory]

Load the bug report:

Figure 5

If the user chooses to view an existing review report, they must select a previously generated report. The use of JSON ensures that the report remains structured and easily readable in various editors, as illustrated in Figure 14.

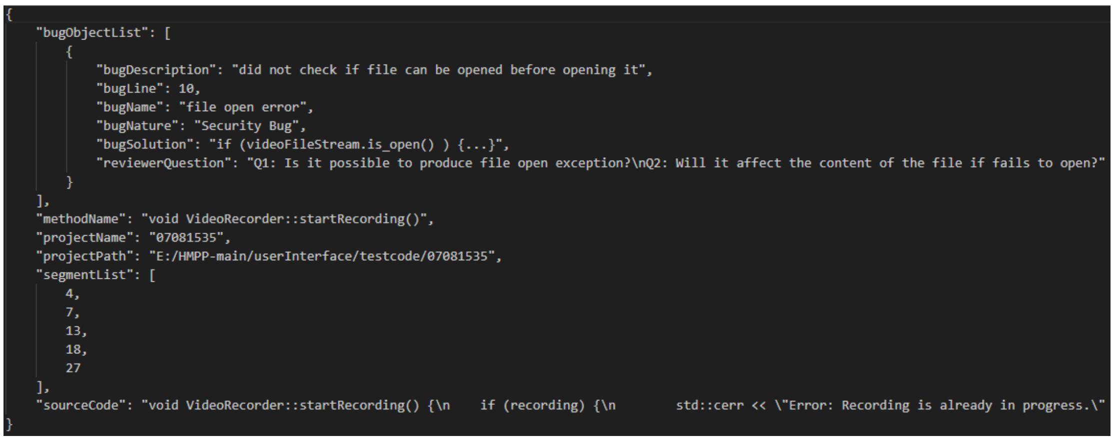
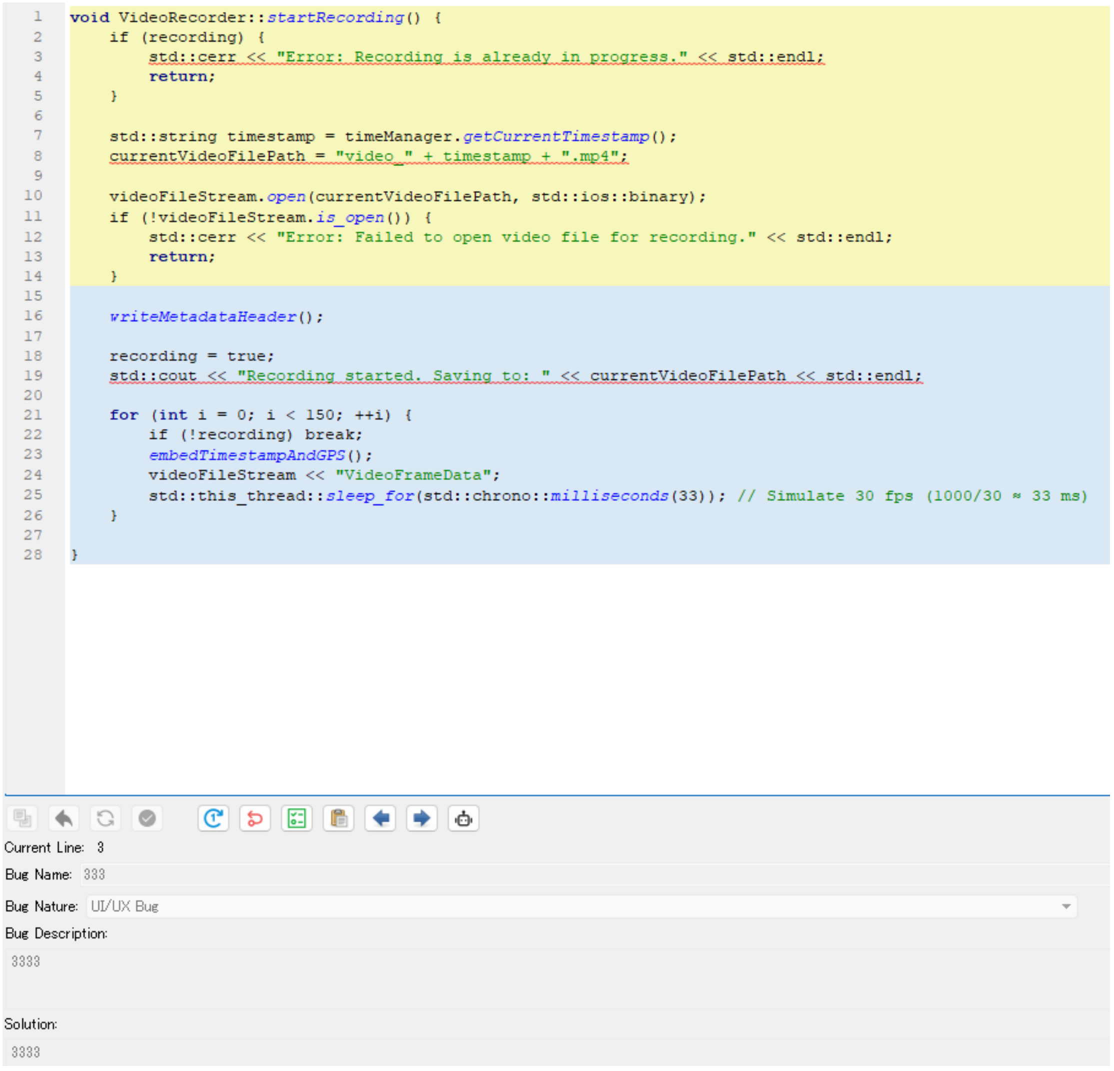

```
{
    "bugObjectList": [
        {
            "bugDescription": "did not check if file can be opened before opening it",
            "bugLine": 10,
            "bugName": "file open error",
            "bugNature": "Security Bug",
            "bugSolution": "if (videoFileStream.is_open() ) {...}",
            "reviewerQuestion": "Q1: Is it possible to produce file open exception?\nQ2: Will it affect the content of the file if fails to open?"
        }
    ],
    "methodName": "void VideoRecorder::startRecording()",
    "projectName": "07081535",
    "projectPath": "E:/HMPP-main/userInterface/testcode/07081535",
    "segmentList": [
        4,
        7,
        13,
        18,
        27
    ],
    "sourceCode": "void VideoRecorder::startRecording() {\n    if (recording) {\n        std::cerr << \"Error: Recording is already in progress.\"
}
```

Figure 14

When opened within the tool itself, lines containing reported bugs are highlighted with red wavy underlines, allowing reviewers to conveniently access all recorded bug information, as shown in Figure 15.



```cpp
void VideoRecorder::startRecording() {
    if (recording) {
        std::cerr << "Error: Recording is already in progress." << std::endl;
        return;
    }

    std::string timestamp = timeManager.getCurrentTimestamp();
    currentVideoFilePath = "video_" + timestamp + ".mp4";

    videoFileStream.open(currentVideoFilePath, std::ios::binary);
    if (!videoFileStream.is_open()) {
        std::cerr << "Error: Failed to open video file for recording." << std::endl;
        return;
    }

    writeMetadataHeader();

    recording = true;
    std::cout << "Recording started. Saving to: " << currentVideoFilePath << std::endl;

    for (int i = 0; i < 150; ++i) {
        if (!recording) break;
        embedTimestampAndGPS();
        videoFileStream << "VideoFrameData";
        std::this_thread::sleep_for(std::chrono::milliseconds(33)); // Simulate 30 fps (1000/30 ≈ 33 ms)
    }

}
```

Current Line: 3

Bug Name: 333

Bug Nature: UI/UX Bug

Bug Description:

3333

Solution:

3333

Figure 15