C++ project: File Handler

Alexandre Quenon

2017-10-16

Contents

1 Functional specifications

Functional specifications define the aims as well as the different functions, or blocks, of the whole system.

1.1 Aims

As the "File_Handler" name suggests, the aim of the current project consists in *handling files*. Handling refers to:

- automatically opening and closing files (RAII[1] class),
- opening either for read or write operations according to the request of the user,
- allow access to only one thread at a time.

TBC! However, there is **no interpretation** of the file content: the "File_Handler" reads and writes but does not understand!

In the case of a multi-thread application, the use of any file stream is equivalent to an access to a *critical resource*. Indeed, if different threads use simultaneously the same stream, a critical race will occur and the data will be mixed in an unpredictable way. In order to prevent such an event from occurring, a control of the resource access must be implemented.

1.2 End-User Interface

The "File_Handler" shall provide two types of interface:

1. read operations,
2. write operations.

Conversely, specific features must absolutely be hidden to the end-user. Hence, they must not be part of the interface.

Expected interface for the read operations:

-

Expected interface for the write operations:

[1]RAII, which stands for "Resource Acquisition Is Initialization", is an object-oriented programming technique which consists in acquiring the resource at object construction and releasing it at destruction, making the resource tied to the object lifetime

-

Hidden features that are not part of the end-user interface:

- resource management (acquisition, request of use and release);
- strategy for thread safe multiple access.

2 Design specifications

Toto

2.1 The “File_Reader” class

xxx

2.1.1 Behaviour at construction

1. try to open the file stream in the specified mode,
2. verify if the file is correctly opened.

2.1.2 Behaviour at destruction

1. close the file stream.

2.1.3 UML class diagram

xxx

2.2 The “File_Writer” class

xxx

2.2.1 Behaviour at construction

xxx

1. try to open the file stream in the specified mode,
2. verify if the file is correctly opened,
3. write a timestamp.

2.2.2 Behaviour at destruction

xxx

1. write a timestamp,
2. close the file stream.

2.2.3 UML class diagram

Methods are organised as follows:

- high-level interface
  - special text formatting
- low-level engine
  - resource acquisition and release
  - basic writing (resource request with mutex)

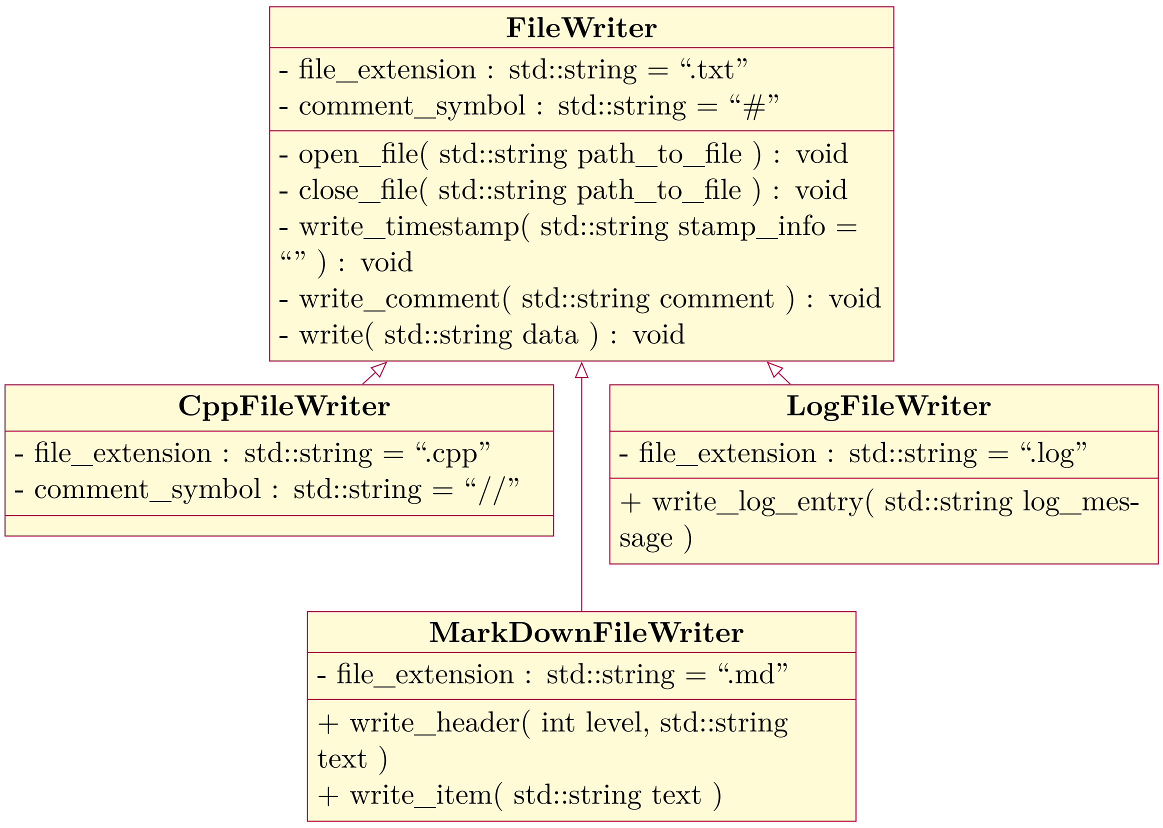
FileWriter
- file_extension : std::string = ".txt"
- comment_symbol : std::string = "#"
- open_file(std::string path_to_file) : void
- close_file(std::string path_to_file) : void
- write_timestamp(std::string stamp_info = "") : void
- write_comment(std::string comment) : void
- write(std::string data) : void
CppFileWriter
- file_extension : std::string = ".cpp"
- comment_symbol : std::string = "//"
LogFileWriter
- file_extension : std::string = ".log"
+ write_log_entry(std::string log_message)
MarkDownFileWriter
- file_extension : std::string = ".md"
+ write_header(int level, std::string text)
+ write_item(std::string text)

Figure 2.1: UML class diagram of the "File_Writer" class