Advanced Prorgamming Concepts

September 7, 2022

Advanced Programming Concepts, assignment 1 (week 1)

In this assignment you'll practice *dependency injection* and the *dependency inversion principle*. All this to make the design of `logger` more robust and flexible.

Problem statement

The `logger` that we developed during this week is far from perfect. It's definition looks simple:

```
class logger: public loggers::ilogger {
public:
    explicit logger(std::ostream& out) noexcept;
    logger() noexcept;
    void log(const std::string& msg) const override;
private:
    std::ostream& m_out;
    void output_time() const;
};
```

Yet, it breaks at least three important software design rules that give three letters to the *SOLID* acronym:

- The Single Responsibility Principle (S)
- The Open-Close Principle (O)
- The Dependency Inversion Principle (D)

Why are they broken?

1. The **SRP** is violated because `logger` not only outputs log messages but is also responsible for time-stamping them.
2. The **OCP** is partially violated because the functionality of `logger` cannot be easily extended without changing it in a way that forces recompilation.
3. The **DIP** is violated because `logger` depends strongly on the `<ctime>` header's functions.

There are also some other minor design issues that need refactoring. The most prominent among them is using `std::string` as the type for messages that are passed for logging. As you'll soon see, there are better options.

When doing this assignment you will:

- Get rid of the `<ctime>` dependency in `logger` by creating a new abstract class `itime_source` and its concrete implementation.
- Get rid of the `std::string` usage in `logger` and `ilogger`

- Get rid of the `ostream` dependency in `logger` by creating a new abstract class `itext_writer` and two implementations of it.

Tasks

`<ctime>` dependency

Start with the biggest offender: the `<ctime>` header usage (dependency) in `logger`. To remove it, you'll need:

- An interface, `itime_source`, that defines a function(s) that returns a timestamp in a **text form**.
- A concrete implementation of this interface (e.g. `system_time_source`), this one can (and possibly must) use `<ctime>` in its implementation file.
- A way to inject the concrete time source object into `logger`–this can be done by:
  - Adding a private pointer member to `logger` (`std::unique_ptr<itime_source>`) that will hold a concrete instance of `itime_source`
  - Adding a function to `logger` for setting the time source.

  If in doubt about how to do it, check how this is done for `program` and its private member `m_logger`.

Notice, that by relegating time-stamping to another class we also fixed the **SRP** issue.

BTW, because `logger` uses streams for output, it might be an idea provide a *stream output operator* in `itime_source`. (But you will have to refactor it later again.)

`std::string` alternatives

By fixing the first problem, we also partially fixed the **OCP** one. Now it's a bit easier to extend `logger` without really modifying it. Time to bring the final touch: `logger` also depends on two other classes: `std::string` and `std::ostream`.

- Investigate `std::string_view`, and replace usage of `std::string` in `logger` with it. For one thing, `std::string_view` is more generic and offers easy conversions from other string types.

`std::ostream` must go

The dependency on `std::ostream` must go. What we need instead is an interface (abstract class) that exposes functions for writing text to some destination and a concrete implementation of this interface.

Define the interface `itext_writer` with at least one stream output operator. (Consider adding other `operator<<` overloads to `itext_writer`, for instance for `const char*`, `char`, `int` and `double`.):

```
struct itext_writer {
    virtual itext_writer& operator<<(std::string_view) = 0;
    virtual ~itext_writer() = default;
};
```

The stream output operator, exposed by this interface should be used by `logger` to send text to an output. This cannot work without a concrete class that implements `itext_writer`, so implement two concrete classes that inherit from `itext_writer`:

- the `console_writer` class that outputs text to the console,
- the `stream_writer` class that writes the output text to a file. The name of the file should be passed to the `stream_writer`'s constructor. Naturally, `stream_writer` should open this file for writing.

A concrete instance of `itext_writer` must be *injected* via a unique pointer in the constructor of `logger`. Like this:

```
class logger: public loggers::ilogger {
public:
    logger(std::unique_ptr<itext_writer> out);
    /* ~~~ */
private:
    std::unique_ptr<itext_writer> m_writer;
    /* ~~~ */
};

int main() {
    // loger is a pointer to a lib::logger object
    // a unique console_writer pointer is passed to the lib::logger's constructor
    auto logger = std::make_unique<lib::logger>( std::make_unique<writers::console_writer>() )

    // or, more verbose, and now with stream_writer for a change
    auto writer = std::make_unique<writers::stream_writer>("log.txt");
    auto logger = std::make_unique<lib::logger>(std::move(writer));
}
```

Final tests

When you've implemented everything, and changed `logger` to use the `itime_source` and `itext_writer` abstractions instead of `<ctime>` and `<iostream>`, test your program in the `main` function. You should at least demonstrate:

- Creating a `logger` instance with one of the writers passed to its constructor.
- Setting the time source of `logger` to `system_time_source`.
- Running the `program` class with the `logger` object passed to it.

Your program should work with either `itext_writer` implementation.