Exercises: SOLID

1. Stream Progress Info

Refactor the code for this task, so that **Stream Progress Info** can work with different kinds of **Streams**. First make sure it works with **Music** too. Refactor the code, so in the future if a **new kind of stream** is introduced, you will need to **just import one new class** with **BytesSent** and **Length** getters in it.

2. Graphic Editor

Refactor the code for this task, so that **Graphic Editor can draw all kinds of shapes** without checking **what kind is the concrete shape.** In the future, new shapes will be added to the system, so prepare the system for those moments. When you **add a new shape**, you should just **add a new class and nothing more**.

3. Detail Printer

Refactor the code for this task, so that **Detail Printer** doesn't need to ask **what kind of an employee is passed to it**. Detail Printer needs to just print details for all kinds of employees. When a new kind of employee is added, you will only need to **add a new class and nothing more.**

4. Recharge

You are given a library with the following classes:

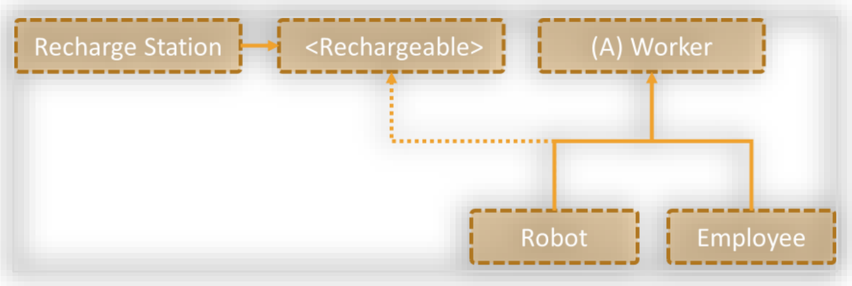
- `Worker implements ISleeper`
- `Employee inherits Worker`
- `Robot inherits Worker`
- `RechargeStation`

If you inspect the code, you can see that some of the classes have methods that they can't use (throw **`UnsupportedOpperationException`**), which is a clear indication that the code should be refactored.

Refactor the structure, so that it conforms to the **Interface Segregation** principle.

Hints

Make the **Robot** extend **`Worker`** and at the same time implement **`Rechargeable.`**



5. Logger

Write a logging library for logging messages. The interface for the end-user should be as follows:

Sample Source Code	Output

`ILayout simpleLayout = new SimpleLayout();`
`IAppender consoleAppender =`
`new ConsoleAppender(simpleLayout);`
`ILogger logger = new Logger(consoleAppender);`
 	
`logger.Error("3/26/2015 2:08:11 PM", "Error parsing JSON.");`
`logger.Info("3/26/2015 2:08:11 PM", "User Pesho successfully registered.");`
	`3/26/2015 2:08:11 PM - Info - User Peter successfully registered.`
	`3/26/2015 2:08:11 PM - Error - Error parsing JSON.`

Logger logs data and time (string) and a message (string).

Library Architecture

The library should have the following components:

- **`Layouts`** - define the format in which messages should be appended (e.g. **`SimpleLayout`** displays logs in the format "**`<date-time> - <report level> - <message>`**")
- **`Appenders` -** responsible for appending the messages somewhere (e.g. **`Console`**, **`File`**, etc.)
- **`Loggers`** - hold methods for various kinds of logging (warnings, errors, info, etc.)

Whenever a logger is told to log something, it calls all of its appenders and tells them to append the message. In turn, the appenders append the message (e.g. to the console or a file) according to the layout they have.

Requirements

Your library should correctly follow all of the **SOLID** principles:

- **Single Responsibility Principle** - no class or method should do more than one thing at once
- **Open-Closed Principle** - the library should be open for extension (i.e. its user should be able to create his own layouts/appenders/loggers)
- **Liskov Substitution Principle -** children classes should not break the behavior of their parent
- **Interface Segregation Principle** - the library should provide simple interfaces for the client to implement
- **Dependency Inversion** - no class/method should directly depend on concretions (only on abstractions)

Avoid code repetition. Name everything accordingly.

Implementations

The library should provide the following ready classes for the client:

- **`SimpleLayout`** - defines the format "**`<date-time> - <report level> - <message>`**"
- **`ConsoleAppender`** - appends a log to the console, using the provided layout
- **`FileAppender`** - appends a log to a file, using the provided layout
- **`LogFile`** - a custom file class, which logs messages in a string builder, using a method **`Write()`**. It should have a **getter** for its **size,** which is the **sum** of the **ascii codes** of all alphabet characters it contains (e.g. a-z and A-Z)

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

Follow us:

- **Logger** - a logger class, which is used to **log messages**. Calls each of its **appenders** when something needs to be logged

Sample Source Code	Output

`var simpleLayout = new SimpleLayout();`
`var consoleAppender = new ConsoleAppender(simpleLayout);`
 	
`var file = new LogFile();`
`var fileAppender = new FileAppender(simpleLayout, file);`
 	
`var logger = new Logger(consoleAppender, fileAppender);`
`logger.Error("3/26/2015 2:08:11 PM", "Error parsing JSON.");`
`logger.Info("3/26/2015 2:08:11 PM", "User Pesho successfully registered.");`
	`<log>`
	`  <date>3/31/2015 5:23:54 PM</date>`
	`  <level>Fatal</level>`
	`  <message>mscorlib.dll does not respond</message>`
	`</log>`
	`<log>`
	`  <date>3/31/2015 5:23:54 PM</date>`
	`  <level>Critical</level>`
	`  <message>No connection string found in App.config</message>`
	`</log>`

The above code should log the messages both on the **`console`** and in **`log.txt`** in the format **`SimpleLayout`** provides.

LogFile

A file should write all messages internally and it should keep information about its size.

Size of a file is calculated by summing ASCII codes of all alphabet characters (a-Z). For example, a file appender with simple layout and a single message **`"3/31/2015 5:33:07 PM - ERROR - Error parsing request"`** has size 2606 (including all characters in PM, ERROR, Error, parsing, request). In case of Xml layout, the file would have size 6632, because of the extra characters within the tags.

Extensibility

The end-user should be able to add his own **layouts/appenders/loggers** and use them. For example, he should be able to create his own **`XmlLayout`** and make the appenders use it, **without directly editing** the library source code.

Sample Source Code	Output

`var xmlLayout = new XmlLayout();`
`var consoleAppender = new ConsoleAppender(xmlLayout);`
`var logger = new Logger(consoleAppender);`
 	
`logger.Fatal("3/31/2015 5:23:54 PM", "mscorlib.dll does not respond");`
`logger.Critical("3/31/2015 5:23:54 PM", "No connection string found in App.config");`
	`<log>`
	`  <date>3/31/2015 5:23:54 PM</date>`
	`  <level>Fatal</level>`
	`  <message>mscorlib.dll does not respond</message>`
	`</log>`
	`<log>`
	`  <date>3/31/2015 5:23:54 PM</date>`
	`  <level>Critical</level>`
	`  <message>No connection string found in App.config</message>`

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

Follow us:

	`</log>`

Report Threshold

Implement a **report level threshold** in all appenders. The appender should append only messages with report level **above or equal to** its report level threshold (by **default** all messages are **appended**). The report level is in the order **Info** > **Warning** > **Error** > **Critical** > **Fatal**.

Sample Source Code	Output

`var simpleLayout = new SimpleLayout();`
`var consoleAppender = new ConsoleAppender(simpleLayout);`
`consoleAppender.ReportLevel = ReportLevel.Error;`
 	
`var logger = new Logger(consoleAppender);`
 	
`logger.Info("3/31/2015 5:33:07 PM", "Everything seems fine");`
`logger.Warning("3/31/2015 5:33:07 PM", "Warning: ping is too high - disconnect imminent");`
`logger.Error("3/31/2015 5:33:07 PM", "Error parsing request");`
`logger.Critical("3/31/2015 5:33:07 PM", "No connection string found in App.config");`
`logger.Fatal("3/31/2015 5:33:07 PM", "mscorlib.dll does not respond");`
	`3/31/2015 5:33:07 PM - Error - Error parsing request`
	`3/31/2015 5:33:07 PM - Critical - No connection string found in App.config`
	`3/31/2015 5:33:07 PM - Fatal - mscorlib.dll does not respond`

Only messages from error and above are appended.

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.
Follow us: