## Questions on Software Concepts and Principles

Use the Software Concepts & Principles from last week to answer these questions. In these questions, *"design concept*" refers to the terminology from week 8, like abstraction or cohesion. "*Design principle"* refers to the design principles from week 8, plus "*Separation of concerns"* can be taken as either a concept or design principle.

In the case of coupling and cohesion, you should write "*high coupling*", "*low coupling*", "*high cohesion*", or "*low cohesion*", not simply coupling or cohesion.

Some of the questions require knowledge of the Python logging facility. This information is given in separate documents.

1. Whenever I modify the code for class A, I have to change something in several other classes, too. This is a symptom of what design concept?

2. The collections.abc package has classes like Iterable, Mapping, and Sized, which define behavior that concrete collection classes should implement. This is an example of what *design concept? (This enables polymorphism but the answer to this question is not polymorphism.)*

3. The work of logging is divided into different components, so that each component is responsible for a different aspect of the logging process. This overall design of the logging facility uses what design principle?

4. Filter, Formatter, and Handler are each responsible for one specific job. This is an example of what design *principle*?

5. When some code invokes logger.info(...), logger.error(...), etc., a **LogRecord** is created. A LogRecord contains the message text, timestamp, and all other info related to the logging event. Packing all this info into a single object is an example of what design concept?

6. A Handler represents a model of something that "handles" log messages, without any specifics or details of *how* the messages are handled. This is an example of what design concept?

7. FileHandler, StreamHandler, etc. all provide the same methods, so Logger can invoke them without knowing which kind of handler is receiving the message. This is what design concept?

8. Logger depends only on the base (model) Handler type, not on any specific kinds of handlers. This is an example of what design principle?

9. Logging is too complicated! We could have the Logger class handle the log messages itself (using some conditional logic) and eliminate all the Handler classes. What *design principle* says that it is better to separate the Handler code from the Logger code?

10. All the *methods* in Filterer are related to filtering messages. This is what design *concept*?

## Design Patterns

1. Logger provides all the methods that the programmer normally invokes to use logging. This avoids the need for a programmer to directly use the different classes in the logging module.

This is an example of what design pattern?

2. RotatingFileHandler adds new behavior to the basic FileHandler, while preserving the same interface. It rotates log files based on either age or file size. For example, if we want to keep 3 recent logs, RotatingFileHandler would create files such as:

messages.log (current log file)

messages.log.1 (previous log file)

messages.log.2 (next previous log file)

messages.log.3 (oldest log file)

RotatingFileHandler uses (or "is") a FileHandler but enhances its behavior. This is what design pattern?

3. Logger inherits from Filterer, which provides these methods:

addFilter(filter)

filter(record: LogRecord) - *the default is to allow the record to be logged. Any filter can veto this and the record is then dropped.*

What design pattern do you think is used by the *filter* method?

Since *filter* a method, it should be a "*Behavioral*" design pattern.