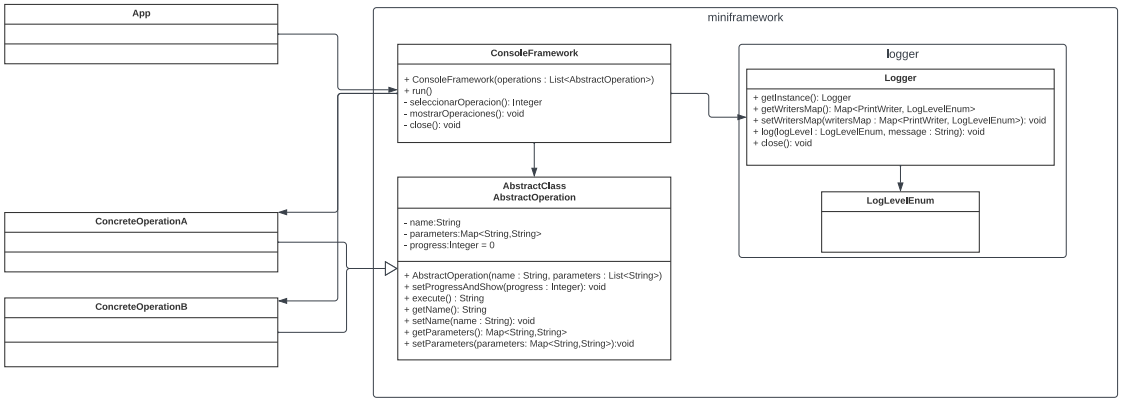
Para el diseño del miniframework empiezo creando la clase AbstractOperation que será el prototipo de las operaciones que crearán los usuarios del framework. Una operación tendrá un nombre que la identifica en el menú que mostrará ConsoleFramework una serie de parámetros, en el constructor se piden en forma de lista y en el objeto se guardarán en forma de mapa donde la clave es el nombre del parámetro introducido en el constructor y el valor se inicializa a cadena vacía. Esta forma deja la responsabilidad de validar los parámetros a la función execute() que implemente el programador.

Una alternativa sería crear una nueva clase parámetro dentro del miniframework que tenga esa responsabilidad, guardándose en la operación un mapa Map<String, Parametro>. El programador tendría que crear una clase Parametro por cada tipo de validación que quisiese.

La clase ConsoleFramework usará una lista de operaciones que captura en el constructor para mostrar el menú con las correspondientes opciones, solicitar los parámetros de la operación seleccionada y llamar al método execute() de la operación para iniciarla.

La clase Logger es una clase Singleton que permite la creación de logs. Almacena una serie de PrintWriter asociados a un nivel de mensaje de log (LogLevelEnum). Al llamar al método log() desde cualquier parte del programa se registrará el mensaje de log según su gravedad en los PrintWriter que correspondan (Consola, Fichero…)



Manual:

Run():

Para ejecutar el framework llamaremos a ConsoleFramework.run() lo cual ocupará el hilo hasta que el usuario seleccione la opción salir o error esperado.

El framework permite añadir operaciones personalizadas, para ello se usará el constructor.

Ej:

List<AbstractOperation> operaciones = new ArrayList<>();

operaciones.add(new OperacionA());

operaciones.add(new OperacionB());

ConsoleFramework myConsoleFramework = new ConsoleFramework(operaciones);

myConsoleFramework.run();

Crear Operaciones:

Para crear Operaciones propias extenderemos AbstractOperation e implementaremos un método execute(): String con la lógica de la Operación, recordar que en esta versión hace falta que el programador valide los parámetros dentro de este método.

EJ:

```
public class SumOperacion extends AbstractOperation {
    public SumOperacion() {
      // nombre de la operacion -- Lista de parametros
        super("Sumar", List.of("Sumando A", "Sumando B"));
    }

    @Override
    public String execute() {
// muestra al usuario que porcentaje de trabajo se ha realizado
        this.setProgressAndShow(0);
        if (getParameters().size() != 2) {
            this.setProgressAndShow(100);
            return "Error: Se requieren dos parámetros para sumar.";
        }
        this.setProgressAndShow(30);

        int sum = 0;
        for (String param : getParameters().values()) {
            try {
                sum += Integer.parseInt(param);
            } catch (NumberFormatException e) {
                this.setProgressAndShow(100);
                return "Error: Uno o más parámetros no son numéricos.";
```

```
            }
            this.setProgressAndShow(60);
        }
        this.setProgressAndShow(100);

        return "Resultado de la suma: " + sum;
    }
}
```

Uso del Logger:

La clase Logger está implementada mediante un singleton, es accesible desde cualquier parte del programa. Para crear un mensaje de log llamaremos al método log() de una instancia de Logger de la siguiente manera:

```
Logger logger = Logger.getInstance();
logger.log(LogLevelEnum.INFO, "Sqrt finalizada");
```

Esta versión permite 3 niveles de gravedad en el log INFO<DEBUG<ERROR. Por defecto Logger guardará cualquier mensaje de log en miniframework.log y lo mostrará por pantalla. Este comportamiento puede ser modificado mediante los siguientes métodos editando el mapa de PrintWriters y sus correspondientes niveles de gravedad.

```
public class Logger {
{…}

public Map<PrintWriter, LogLevelEnum> getWritersMap();
public void setWritersMap(Map<PrintWriter, LogLevelEnum> writersMap);
{…}

}
```