SEMINAR

4 Logger

(cont.)

Diseño del Software

Grado en Ingeniería Informática del Software

2024-2025

Escuela de Ingeniería Informática
Universidad de Oviedo

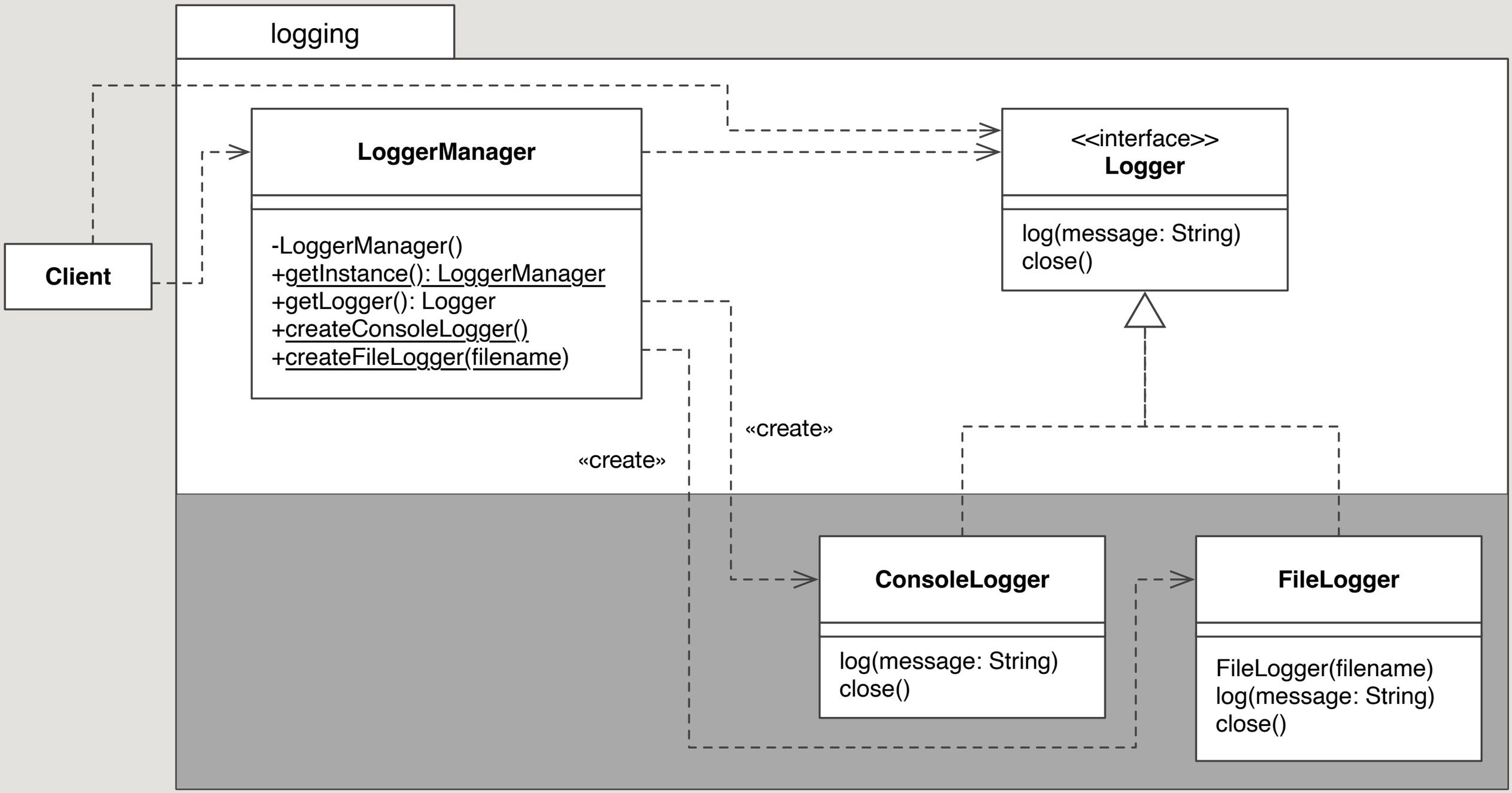
César Acebal

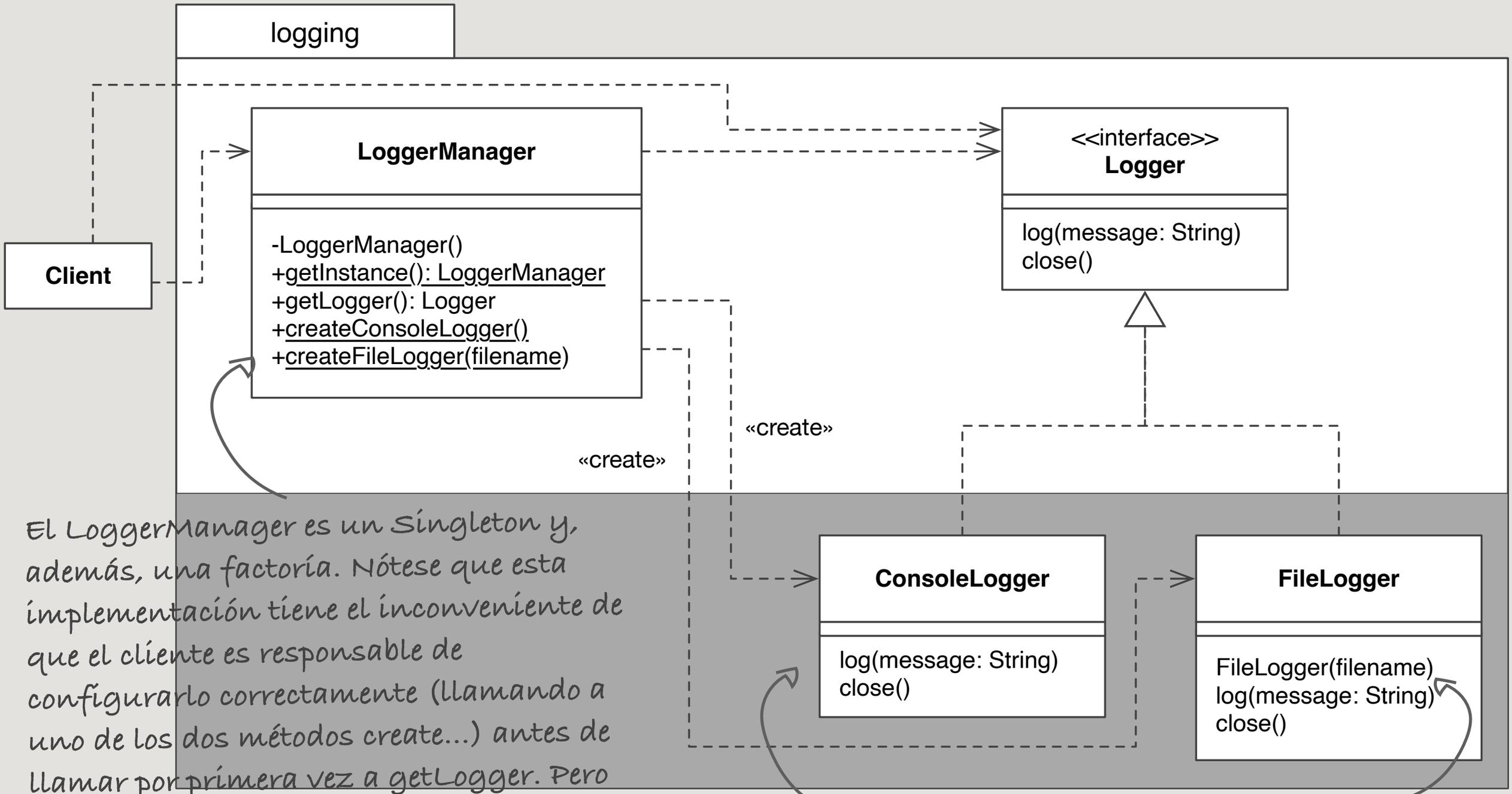
Habíamos quedado aquí.

Dos tipos: de consola y fichero.

Una vez decidido el tipo de logger y creado este, ya no se podrá cambiar.

Se mantiene el requisito de que la instancia del logger creado sea única durante toda la ejecución del programa.


logging
LoggerManager
-LoggerManager()
+getInstance(): LoggerManager
+getLogger(): Logger
+createConsoleLogger()
+createFileLogger(filename)
<<interface>>
Logger
log(message: String)
close()
Client
«create»
«create»
«create»
ConsoleLogger
log(message: String)
close()
FileLogger
FileLogger(filename)
log(message: String)
close()



El LoggerManager es un Singleton y, además, una factoría. Nótese que esta implementación tiene el inconveniente de que el cliente es responsable de configurarlo correctamente (llamando a uno de los dos métodos create...) antes de llamar por primera vez a getLogger. Pero también se podría haber leído un fichero de propiedades durante la inicialización estática de la clase.

A ConsoleLogger y FileLogger podemos darles visibilidad de paquete para que no sean accesibles desde fuera del paquete «logging»; si no, tendríamos que hacer sus constructores no públicos, para evitar que se pudiesen crear objetos directamente de ellos.

¿Es LoggerManager un patrón de diseño?

No, se trata de una factoría simple.

Recordad: llamamos así a cualquier clase que crea y devuelve objetos de otras clases.

¿Es necesaria?

No; podría encargarse la propia clase Logger.

Aquí lo hacemos así para abstraer la lógica de creación. Ayuda a cumplir el principio de responsabilidad única.

Configuración

¿Qué enfoques hay para crear inicialmente el logger del tipo adecuado?

Un método de creación por cada tipo de logger

LoggerManager.java

```java
public static void createConsoleLogger() {
    checkNoLogger();
    logger = new ConsoleLogger();
}

public static void createFileLogger(String fileName)
        throws IOException {
    checkNoLogger();
    logger = new FileLogger(fileName);
}
```

Una variante del anterior

Un método de creación parametrizado

```
public static void createLogger(...) { ... }
```

Tiene el inconveniente de que cada tipo de logger puede necesitar distintos parámetros.

Leer de un fichero de configuración

Logger.java

```java
Properties properties = new Properties();
properties.load(new FileInputStream("./logging.properties"));
String loggerType = properties.getProperty("loggerType");

if (loggerType.equals("console")) {
    logger = ConsoleLogger(PLAIN_TEXT_FORMAT);
} else if (loggerType.equals("file")) {
    String filename = properties.getProperty("filename");
    logger = FileLogger(filename, PLAIN_TEXT_FORMAT);
}
// ...
```

Un tercer tipo de logger

Se produce un nuevo cambio en los requisitos: ahora nos piden que haya también un logger en formato HTML.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>Logger</title>
</head>
<body>
    <h1>Logger</h1>
    <ul>
        <li>Hello, world!</li>
        <li>Second message</li>
        <li>Another...</li>
    </ul>
</body>
</html>
```

¿Cómo lo haríais?

A hint

Identificar los aspectos de la aplicación que cambian y separarlos de lo que tiende a permanecer igual.

O, lo que es lo mismo... encapsular y aislar el concepto que varía.

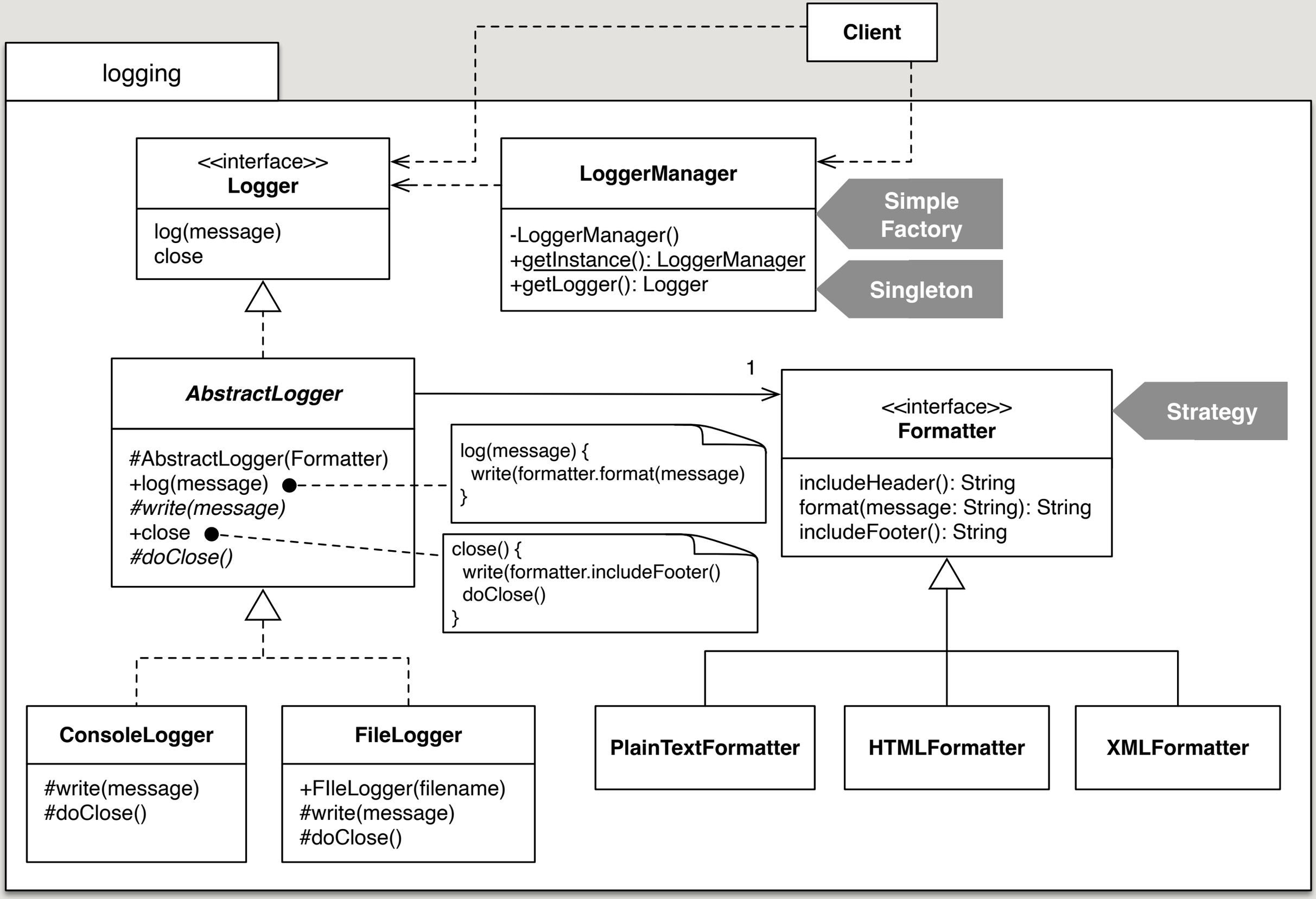
¿Qué puede variar aquí?

¿Qué puede variar aquí?

¿No podrían surgir nuevos tipos de logger que escribiesen en una base de datos relacional, o en un servidor remoto, a través de la red?

¿Y otros formatos como XML?

Más aún: ¿no podría tener sentido tener un logger de base datos pero en HTML? ¿O en XML?


logging
Client
<<interface>>
Logger
log(message)
close
LoggerManager
-LoggerManager()
+getInstance(): LoggerManager
+getLogger(): Logger
Simple Factory
Singleton
AbstractLogger
#AbstractLogger(Formatter)
+log(message)
#write(message)
+close
#doClose()
1
log(message) {
write(formatter.format(message)
}
close() {
write(formatter.includeFooter()
doClose()
}
<<interface>>
Formatter
includeHeader(): String
format(message: String): String
includeFooter(): String
Strategy
ConsoleLogger
#write(message)
#doClose()
FileLogger
+FIleLogger(filename)
#write(message)
#doClose()
PlainTextFormatter
HTMLFormatter
XMLFormatter

Estábamos mezclando dos conceptos independientes:*

El medio de salida

Consola, fichero, BD, internet…

El formato

Sin formato, HTML, XML…

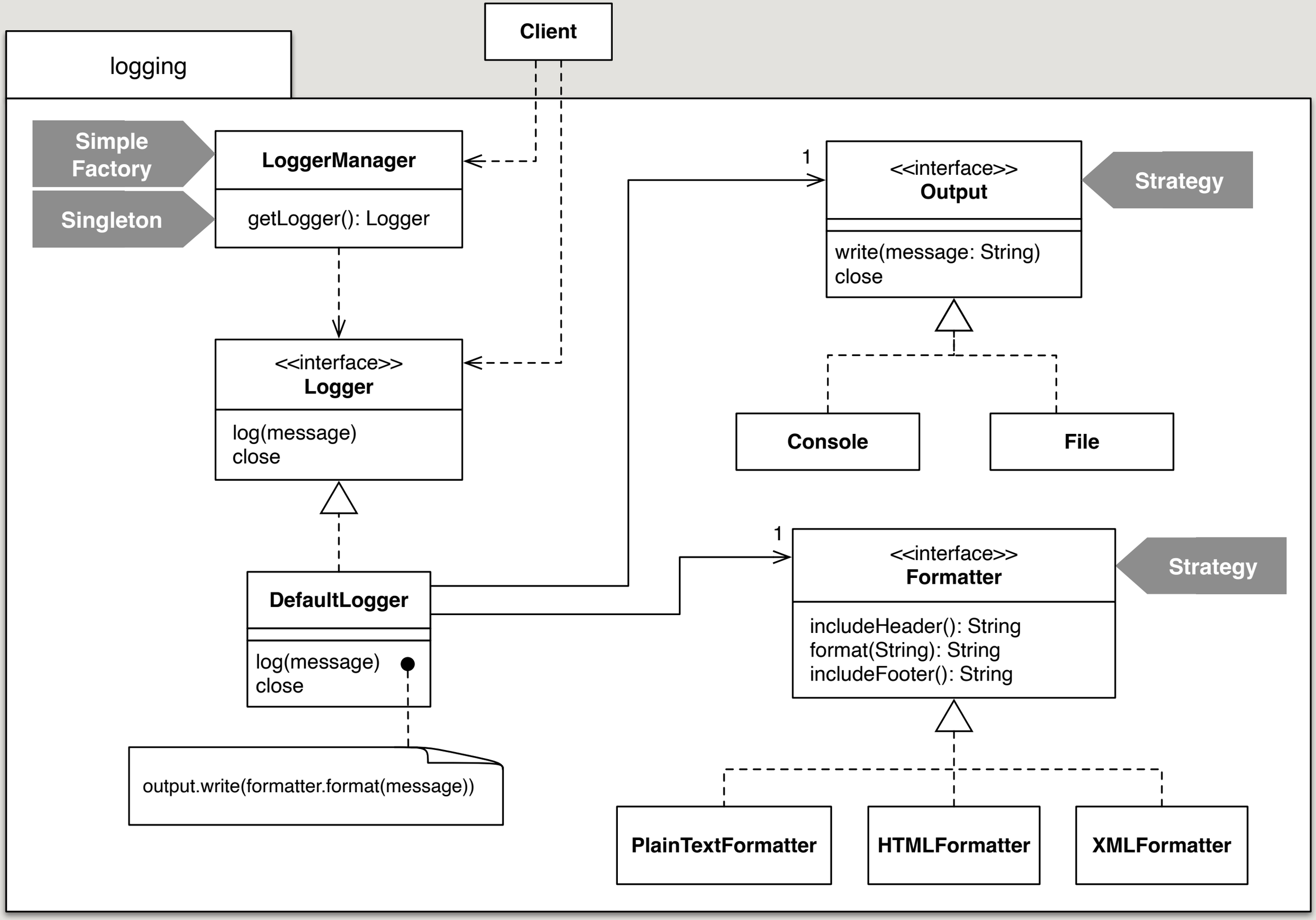
* Que pueden cambiar por diferentes motivos.

¿Y qué pasa con el medio de salida?

¿No se podría sacar fuera también, tal como hemos hecho con el formato?

(En vez de usar la herencia).

Sí, desde luego, sería otra posibilidad.


logging
Client
Simple Factory
Singleton
LoggerManager
getLogger(): Logger
<<interface>>
Logger
log(message)
close
DefaultLogger
log(message)
close
output.write(formatter.format(message))
1
<<interface>>
Output
write(message: String)
close
Strategy
Console
File
1
<<interface>>
Formatter
includeHeader(): String
format(String): String
includeFooter(): String
Strategy
PlainTextFormatter
HTMLFormatter
XMLFormatter

Ahora podéis echar un vistazo al módulo de «logging» de la API de Java y ver cómo está diseñado.