Frameworks

Un framework es una aplicación “semi-completa”, “reusable”, que puede ser especializada para producir aplicaciones a medida... un conjunto de clases concretas y abstractas, relacionadas (por herencia, conocimiento, envío de mensajes) para proveer una arquitectura reusable que implementa una familia de aplicaciones (relacionadas)...

- Framework controla la ejecución (execution thread)
    - Inversión de control
- Cookbook: reglas de uso
    - Instanciación: implementar aplicación
        - White-box: thread incompleto
        - Black-box: thread configurable
    - Extensión: agregar opciones

Servidores TCP

Es un programa que implementa una de las partes de la arquitectura Cliente/Servidor.

- Un Servidor escucha en un socket TCP (IP_address+port)
- Un Cliente establece una conexión con el servidor para enviar mensajes (plain text)
- El Servidor responde

Ejemplos:

- EchoServer: Recibe un mensaje de texto y Responde el mensaje que el cliente envió
- SMTPServer: emails
- DayTimeServer: fecha actual
- HTTPServer: envío y recepción de documentos web

Flujo de control

1. Crear un socket (IP_address + port)
2. Escucha en el socket
3. Aceptar al cliente ⇒ Sesión
4. Recibir un mensaje
    a. Condición ⇒ *Responder*
    b. !Condición ⇒ Cerrar sesión e ir a paso 2

La idea es reusar el “flujo de control”. Este está incompleto (generalmente no compila), y debemos hacer que “responda”, es decir “enganchar” (usando *hooks obligatorios*) la funcionalidad.

Inversión de control

Es un principio de diseño donde el control del flujo de un programa se delega a un componente externo, como un framework, en lugar de que el código de la aplicación lo gestione directamente. Esto significa que el framework asume la

responsabilidad de crear, configurar y gestionar los objetos (o componentes) y sus interacciones, en lugar de que el desarrollador lo haga manualmente.

Hotspots vs FrozenSpot

FrozenSpot: aspecto del framework que afecta a todas las instanciaciones y que no se puede modificar (marca indeleble)

- Comportamiento invariable en un framework orientado a objetos

HotSpot: estructura en el código que permite modificar el comportamiento del framework, para instanciar y para extender. Hooks Methods

- Punto de variabilidad de un framework orientado a objetos

Framework de Caja Blanca (white box framework)

- La instanciación hereda y completa el loop de control agregando código
  - Ejercitando un hotspot con herencia
  - Modificando código fuente del framework
  - Es posible que requiera agregar métodos a clases del framework
- Demanda conocimiento del código del framework ⇒ es una Caja Blanca
- Loop de control suele ser Template Method

Framework de Caja Negra (black box framework)

- La instancia se obtiene mediante composición y completa el loop de control agregando código
  - Usando interfaces o clases proporcionadas por el framework
  - Configurando componentes sin modificar el código fuente
  - No requiere herencia directa ni cambios en el framework
- Demanda conocimiento limitado del código del framework ⇒ es una Caja Negra
- Loop de control suele ser manejado por un contenedor o configurador (no Template Method)

Plantillas y ganchos

- Las plantillas y los ganchos se pueden utilizar en un framework para separar lo que se mantiene constante (frozenspot) de lo que varía (hotspots).
- Las plantillas implementan lo que se mantiene constante, los ganchos lo que varía.
- Se puede utilizar herencia o composición para separar la plantilla de los ganchos.
- Si uso herencia, la plantilla se implementa en una clase abstracta y los ganchos en sus subclases (como el clásico patron Método Plantilla). Si uso composición, un objeto implementa la plantilla y delega la implementación de los ganchos en sus partes.

Plantillas y herencia

El patron de diseño "metodo plantilla" documentado en el libro "Patrones de diseño" (de Gamma y otros) propone abstraer los pasos canónicos de la operación en un método implementado en la clase abstracta y redefinir los pasos variables en sus subclases. De esta forma, la herencia es el mecanismo principal para separar lo que es constante, de lo que varía. El método plantilla es parte del frozenspot mientras que las operaciones abstractas (o los ganchos) dan lugar a la implementación de los hotspots.

- Es más simple para casos con pocas alternativas y/o pocas combinaciones
- Al implementar los métodos gancho puedo utilizar las variables de instancia y todo el comportamiento heredado de la clase abstracta
- Si hay muchas variantes o combinaciones, empiezo a tener muchas clases y duplicación de código

Plantillas y composición

La otra estrategia es separar lo constante de lo que cambia componiendo objetos. La separación de lo que es constante y lo que varía no solo se separa en métodos diferentes, sino que también se encapsula en objetos diferentes.
El método plantilla ya no envía mensajes al objeto mismo (como lo hacía en el caso de herencia) sino que delega las operaciones en sus partes. Entonces, la plantilla está en el robot, y los ganchos en las partes.

- Evita la duplicación de código y el creciente numero de clases cuando existen muchas alternativas y combinaciones posibles
- Al implementar los métodos gancho debo pasar como parámetro todo lo que necesiten - no tienen acceso a las variables de instancia del objeto
- Puedo cambiar el comportamiento en tiempo de ejecución sin mayor dificultad

SimpleThreadTCPServer (whitebox)

Extensión

1. No existen puntos de extensión

Ejemplos

- VoidServer
- EchoServer

Instanciación (Cookbook)

1. Subclasificar SimpleThreadTCPServer
    a. Debe implementar Main(String[])
        i. crear una instancia
        ii. enviar método startLoop(String[])
    b. Debe implementar handleMessage(String)
    c. Métodos que podría implementar
        i. acceptAndDisplaySocket(ServerSocket).
        ii. checkArguments(String[])
        iii. displayAndExit(int)
        iv. displaySocketData(Socket)
        v. displayUsage()

vi. displaySocketInformation()
vii. displayWarning()

```
12      public final void startLoop(String[] args) {
13          checkArguments(args);
14
15          int portNumber = Integer.parseInt(args[0]);
16
17
18          try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
19              displaySocketInformation(portNumber);
20              while (true) {
21                  Socket clientSocket = acceptAndDisplaySocket(serverSocket);
22                  handleClient(clientSocket);
23              }
24          } catch (IOException e) {
25              displayAndExit(portNumber);
26          }
27      }
```

Todos los servidores pueden redefinir: checkArguments(), displaySocketInformation() y displayAndExit()

SimpleThreadTCPServer.handleClient(Socket)

```
62      private final void handleClient(Socket clientSocket) {
63
64          try (
65                  PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
66                  BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));) {
67              String inputLine;
68              while ((inputLine = in.readLine()) != null) {
69                  System.out.println("Received message: " + inputLine + " from "
70                          + clientSocket.getInetAddress().getHostAddress() + ":" + clientSocket.getPort());
71                  handleMessage(inputLine, out);
72                  if (inputLine.equalsIgnoreCase("")) {
73                      break; // Client requested to close the connection
74                  }
75              }
76              System.out.println("Connection closed with " + clientSocket.getInetAddress().getHostAddress() + ":"
77                      + clientSocket.getPort());
78          } catch (IOException e) {
79              System.err.println("Problem with communication with client: " + e.getMessage());
80          } finally {
81              try {
82                  clientSocket.close();
83              } catch (IOException e) {
84                  System.err.println("Error closing socket: " + e.getMessage());
85              }
86          }
87      }
88  }
```

Todos los servidores deberán definir: handleMessage(String, PrintWriter)

EchoServer.java
SingleThreadTCPServer (hotspot herencia)

```java
import java.io.PrintWriter;

public class EchoServer extends SingleThreadTCPServer {

    public void handleMessage(String message, PrintWriter out) {
        out.println(message);
    }

    Run | Debug
    public static void main(String[] args) {

        new EchoServer().startLoop(args);

    }
}
```

1. startLoop(args) arranca el loop de control del framework
    a. El Framework toma el control (**inversión de control**)
    b. Ahora está completo porque se implementa handleMessage(...)
2. handleMessage(...) es invocado desde alguna parte del framework
    a. Completa el loop de control
    b. “Hollywood Principle”: no nos llames, nosotros te contactaremos
    c. EchoServer hereda el loop de control no existe encapsulamiento ⇒ es una **Caja Blanca**
    d. La ejecución del server depende de la correcta implementación de handleMessage() porque es parte del loop (incompleto) de control

tcp.server.reply (blackbox)

Cookbook

1. En un objeto “contexto”
    a. instanciar un MessageHandler (puede ser Strategy o Command)
        i. Echo,
        ii. Void
    b. Instanciar ConnectionHandler con el MessageHandler
        i. SimpleConnectionHandler
        ii. MultiConnectionHandler
    c. Instanciar TCPControlLoop con ConnectionHandler
    d. Enviar método startLoop() al TCPControlLoop

```java
import tcp.server.reply.*;

public class EchoApp  {

    Run | Debug
    public static void main(String[] args) {

        new TCPControlLoop(new SingleConnectionHandler(new EchoHandler())).startLoop(args);

    }
}
```

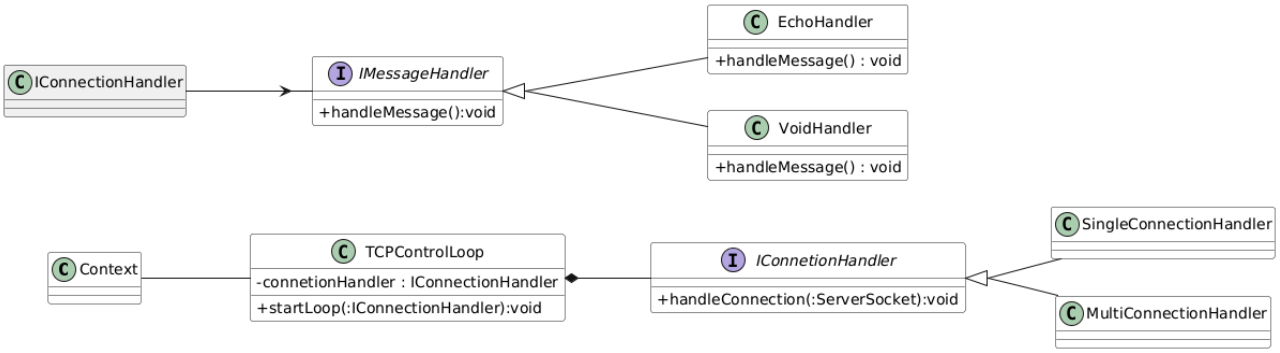
tcp.server.reply
(hotspot composición)

```java
import tcp.server.reply.*;

public class MultiEchoApp  {

    Run | Debug
    public static void main(String[] args) {

        new TCPControlLoop(new MultiConnectionHandler(new EchoHandler())).startLoop(args);
    }
}

```

tcp.server.reply
multisession, echo handler

```java
import tcp.server.reply.*;

public class TestApp  {

    Run | Debug
    public static void main(String[] args) {

        new TCPControlLoop().startLoop(args);

    }
}
```

tcp.server.reply
singlesession, void handler



Context no es parte del framework

FrozenSpot:

- Condición de corte de la conección
- Un tipo de conexión (runtime)
- Un tipo de MessageHandler (runtime)

HotSpot:

- Nuevos MessageHandlers s/funcionalidad: Hash, timestamp, etc
- Nuevos ConnectionHandlers: Timeout, recording, etc

wrap up

- TCPControlLoop se configura con TCPConnection y MessageHandler
- El contexto puede ser:
    - servidor
    - parte de una aplicación
- Posibles mejoras:
    - 1. Crear una Superclase de SingleConnectionHandler y MultiConnectionHandler
    - 2. Crear la jerarquía de EndSessionPolicy
    - 3. Modelar el concepto de Session
        - a. En SingleConnectionHandler la sesión es el loop que procesa mensajes

- b. En MultiConnectionHandler la sesión es el TCPWorker (subclase de Thread)

java.util.logging

- Los logs son útiles para desarrolladores, administradores y usuarios
- Define jerarquía de “labels”
- Activar/Desactivar logs (sin tocar código)
- Generar reportes en varios formatos (txt, json,xml) y destinos (file, screen, socket)
- La aplicación
  - Configura al framework
  - Manda mensajes a objetos Logger
- El Framework se encarga de:
  - Cómo se crean, organizan y recuperan esos objetos
  - Cómo se configuran
  - Cómo se activan y desactivan
  - A qué prestan atención y a qué no
  - Cómo se formatean los logs
  - A dónde se envían los logs

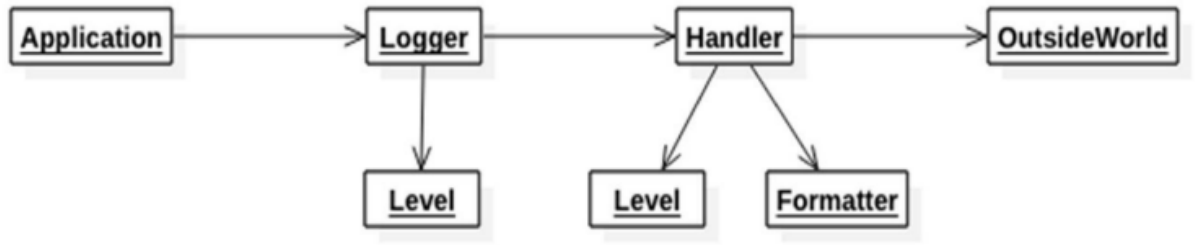
```java

import java.util.logging.Logger;

public class SimpleLoggingExample {

    private static final Logger logger = Logger.getLogger(SimpleLoggingExample.class.getName()

    public static void main(String[] args) {
        logger.info("Application started");

        try {
            int result = 10 / 0; // Simulate an error
        } catch (ArithmeticException e) {
            logger.severe("An error occurred: " + e.getMessage());
        }

        logger.info("Application finished");
    }
}
```

Variante

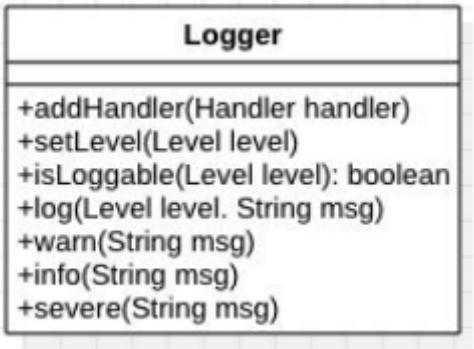
```java
public class Sandbox {
    public static void main(String[] args) throws IOException {
        Logger.getLogger("app.main").addHandler(new FileHandler("log.txt"));
        Logger.getLogger("app.main").log(Level.INFO, "App iniciada");
        try {
            // Acá que hace algo que "podría" resultar en una excepción
            int explodesForSure = 1 / 0;
        } catch (Exception ex) {
            Logger.getLogger("app.main").log(Level.SEVERE, "Explotó!", ex);
        }
        Logger.getLogger("app.main").log(Level.INFO, "App terminada");
    }
}
```

Logger
- mantiene un “registry” de sus instancias.
- getLogger() es un lazy-initializer

Arquitectura visible

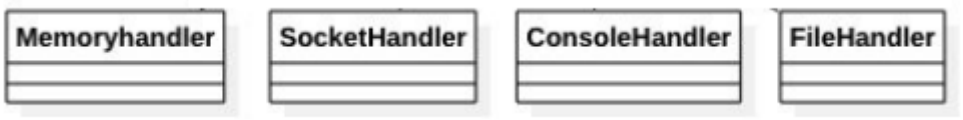


Logger: objeto al que le pedímos que emita un mensaje de log



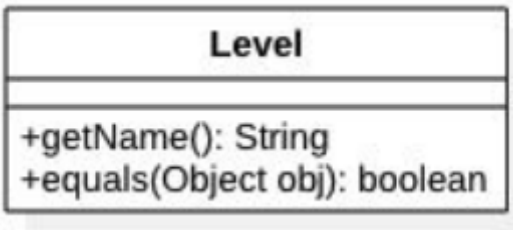
- Podemos definir tantos como necesitemos
    - Instancias de la clase Logger
    - Las obtengo con Logger.getLogger(String nombre)
- Cada uno con su filtro y handler/s
- Se organizan en un árbol (en base a sus nombres)
    - Heredan configuración de su padre (handlers y filters)
- log(Level, String) agrega un mensaje al log
    - Alternativamente uso warn(), info(), severe() ...

Handler: recibe los mensajes del Logger y determina como “exportarlos”



- Puede filtrar por nivel
- Tiene un Formatter

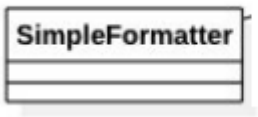
Level: indica la importancia de un mensaje

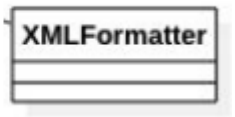


- Cada vez que pido que se loggee algo, debo indicar un nivel
- Los Loggers y Handler comparan el nivel de un cada mensaje con el suyo para decidir si les interesa o no

- Si te interesa un nivel, también te interesan los que son más importantes que ese
- Hay niveles predefinidos, en variables estáticas de la clase Level (SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINIEST, OFF)

Formatter: determina cómo se ”presentará” el mensaje





- El Formatter recibe un mesaje de log (un objeto) y lo transforma a texto
- Son instancias de: SimpleFormatter o XMLFormatter
- Cada handler tiene su formatter
    - Los FileHandler tienen un XMLFormatter por defecto
    - Los ConsoleHandler tienen un SimpleFormatter por defecto

Extendiendo el framework

Y, mirando adentro, puedo agregar nuevas clases de Formater, Handler y Filter

- Nuevo Formatter: Subclasifico la clase abstracta Formatter o alguna de sus subclases
- Nuevo Handler: Subclasifico la clase abstracta Handler o alguna de sus subclases
- Nuevo Filter: Implemento la interfaz Filter