Frameworks

Artefactos de software, con el principal objetivo de **reúso** de código.

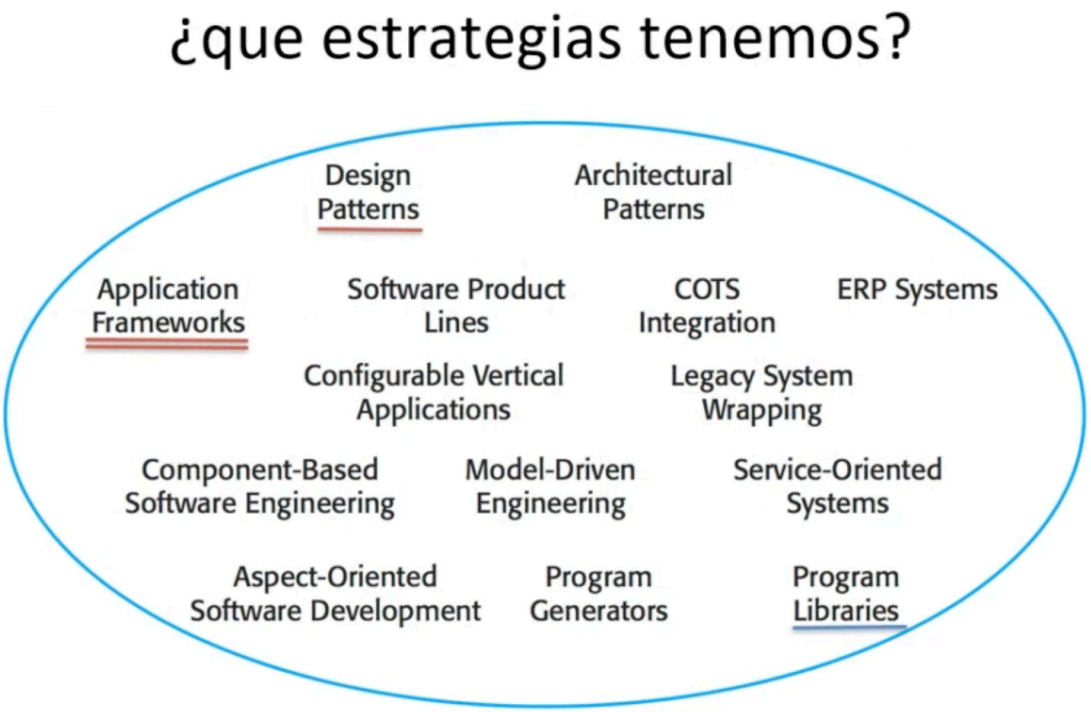
Cuando desarrollamos software, no implementamos desde cero toda la funcionalidad, muchas partes ya están hechas y las puedo reutilizar. Esto incluye código, diseño o estrategias de diseño (patrones de diseño), funciones y estructuras de datos, componentes y servicios (API), o aplicaciones completas que adaptamos e integramos.

¿Por qué reutilizar?

- Reducir costos de desarrollo y mantenimiento.
- Reducir los tiempos de entrega y puesta en el mercado.
- Agilidad de mercado y customización en masa.
- **Para aumentar la calidad**
    - El reúso aprovecha mejores y correcciones. El tiempo da madurez y corrección.
    - Se aprovecha el conocimiento de los especialistas (encapsulado en componentes reusables)

¿Qué dificultades tenemos?

- Problemas de mantenimiento si no tenemos control de los componentes que reusamos.
- Algunos programadores prefieren hacer todo ellos mismos. No es lo mejor.
- Hacer software reusable es más difícil y costoso (paga a la larga, procesos específicos).
- Encontrar, aprender a usar y adaptar componentes reusables requiere de esfuerzo adicional.



Librerías de Clase - Toolkits

- Resuelven problemas comúnes a la mayoría de las aplicaciones.
  - EJ: Manejo de archivos, funciones aritméticas, etc.
- Cada clase en la librería resuelve un problema concreto, es independiente del contexto de uso. No espera nada de nuestro código.
- Nuestro código controla/usa a los objetos de las librerías.

Ejemplos típicos son java.util.collections , Math.

Frameworks orientados a objetos

Una framework es una aplicación “semicompleta”, “reusable” que puede ser especializada para producir aplicaciones a medida.

El artefacto resuelve gran parte de la implementación que tengo que hacer. Cuando creo una página web con Django tengo que completar las partes que queden para que funcione, porque ya el framework me da gran parte del manejo.

A veces un framework sólo no me alcanza y tengo varios trabajando en conjunto.

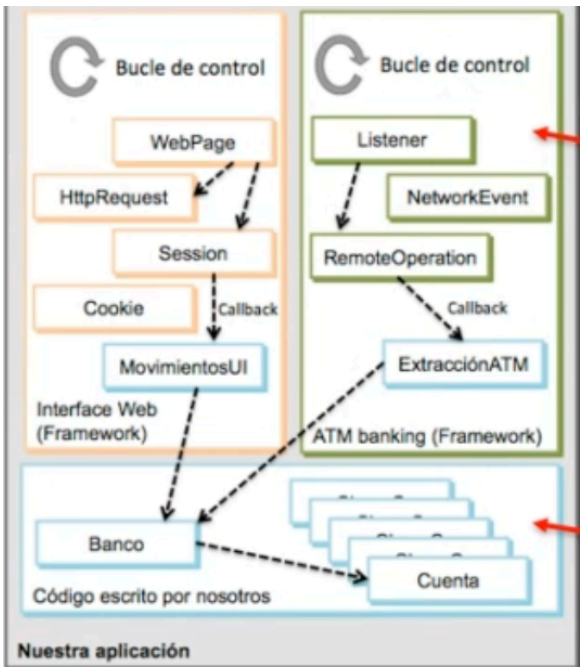
Un framework es un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas. Sus clases trabajan en conjunto, no de forma aislada.

Los desarrolladores incorporan el framework en sus aplicaciones y lo especializan para cubrir sus necesidades.

- El framework define el esqueleto y el desarrollador define sus propia características para completar ese esqueleto.
- A diferencia de una librería, un framewoek provee una estructura coherente en lugar de un conjunto de clases útiles.
- Las clases en el framework se relacionan (herencia, conocimiento, envío de mensajes). Conforman un todo.

Inversión de control: El código del framework controla al mío.

- Estamos completando el código que hizo otro. A veces puedo ver el código fuente y a veces no.



Frameworks de infraestructura

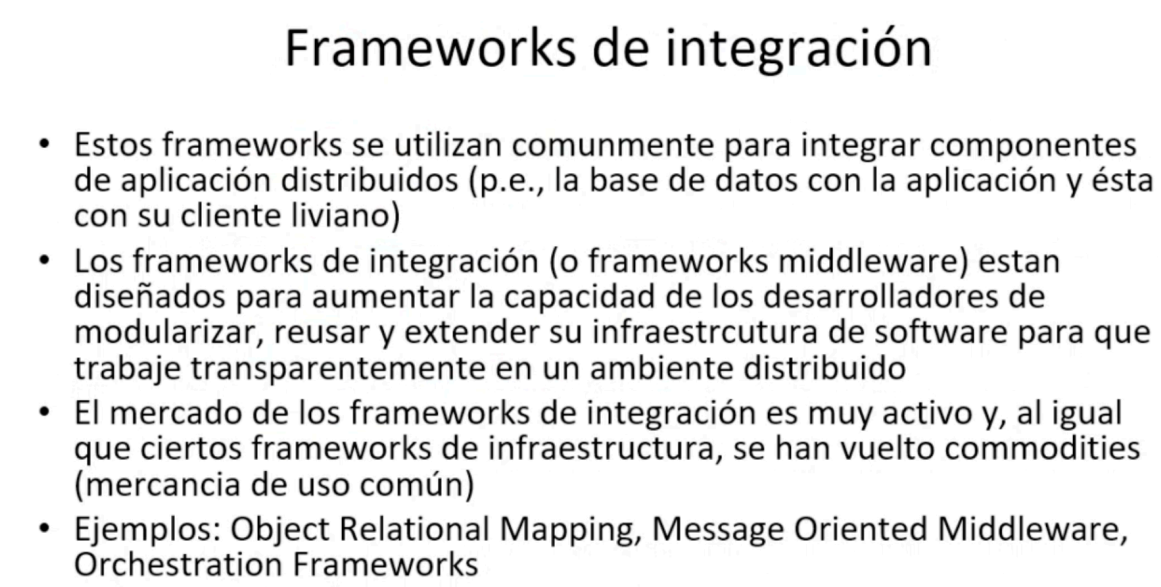
Ofrecen una infraestructura portable y eficiente sobre la cual construir una gran variedad de aplicaciones.

Algunos de los focos de este tipo de framworks son:

- interfaces de usuario (desktop, web, móviles)
- seguridad
- contenedores de aplicación
- etc

Atacan problemas generales del desarrollo de software, que por lo general no son percibidos completamente por el usuario de la aplicación (resuelven problemas de los programadores, no de los usuarios).

Es común que sean incluídos como parte de plataformas de desarrollo (Java tiene, .NET también).



Frameworks de aplicación

Estos frameworks atacan dominios de aplicación amplios que son pilares fundamentales de las actividades de las empresas.

Ejemplos de estos son aquellos en el dominio de los ERP (Enterprise Resourve Planning), CRM (Customer Relationship Managment), Gestión de documentos, Cálculos financieros.

Los problemas que atacan derivan directamente de las necesidades de los usuarios de las aplicaciones y por lo tanto hacen que el retorno de la inversión en su desarrollo/adquisición sea más evidente y justificado.

Un framework enterprise puede encapsular conocimiento y experiencia de muchos años de una empresa, transformándose en la clave de su competitividad y es su más preciado capital.

Capturan el conocimiento de una gran cantidad de clientes de una empresa (por ej), lo que le permite a la empresa ser más productiva. Agrupan el manejo común y especializan la parte específica de cada aplicación. Cuantos más clientes y más aplicaciones, más calidad y más valor tiene dicho framework.

Un ejemplo de este es spring.

Frozenspot vs Hotspot

Estos conceptos tienen que ver más con la especificación de requerimiento, no de una clase en específica.

Froezenspots de un framework

- Todas las aplicaciones construídas con un mismo framework tienen aspectos en común que no podemos cambiar.
- Es la parte del framework que no puede cambiar o que cambia muy de vez en cuando.

Hotspots de un framework

- El framework ofrece puntos de extensión que nos permiten introducir variantes y así construir aplicaciones diferentes.
- La parte del framework que cambiamos.

Caja blanca vs Caja negra

Los puntos de extensión pueden implementarse en base a herencia o en base a composición.

- A los frameworks que utilizan herencia en sus puntos de extensión, les llamamos de Caja Blanca (WhiteBox).
  - Me tengo que meter más en la implementación del framework.
  - Más difícil de usar(tengo que entender cómo funciona), pero más flexible.
- A los que utilizan composición les llamamos de Caja Negra (Blackbox).
  - Con este tipo no me interesa conocer cómo funciona el framework por dentro.
  - Más fácil de usar(no tengo que entender cómo funciona internamente), pero menos flexible.
- La mayoría de los framewoeks están en algún punto en el medio:
  - Algunos usuarios los ven como caja negra.
  - Otros usuarios lo ven como caja blanca.

---

java.util.logging

Ejemplo de una empresa que desarrolla un sistema para flota de vehículos. Agrupa el comportamiento común en un framework y luego puede reutilizarlo para realizar un sistema de flota de taxis, de micros, etc (frozenspot).

Cuando tiene que desarrollar un nuevo sistema de flota hace uso del framework y completa las partes que faltan para ese tipo específico (hotspot). Es decir, completa esos “huecos” que faltan para que funcione.

Inversión de control: Sucede cuando el código del framework llama al código que agregué. Característica frecuente en los frameworks.

Logging - Propósito

Se agrega código de logging a la aplicación para entender lo que pasa con ella:

- Reportes de eventos importanes, errores y excepciones.

- Pasos críticos en la ejecución.
- Inicio y fin de operaciones complejas o largas.

Los logs son útiles para desarrolladores, administradores y usuarios. No reemplazan al testing.

> Tiene el mismo propósito (o similar), a cuando se agregan prints en el medio del código de un programa, para poder ir viendo qué es lo que está haciendo que falle. Como un debugger, pero para cuando el programa está corriendo en un servidor.

Herramientas y Estrategias

Se podría usar System.out.println , pero los framewoeks/APIs de loggin permiten:

- Estandarizar la práctica.
- Activar o desactivar logs selectivamente, incluso sin modificar el código.
  - No tengo que agregar manualmente los print y luego sacarlos. Lo activo y desactivo cuando quiero, y no tengo que modificar en nada el código.
- Enviar los reportes a distintos formatos y destino (txt, JSON, XML, archivos, etc).
  - Ej: Que se envíen una vez al mes a un correo.
- Oculta los detalles de implementación.

Java Logging FrameWoek

Framework de logging, incluído en el SDK de Java. Define:

- El log se hace enviando mensajes a objetos Logger.
- 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
- A dónde se envián.

—

- El framework se encarga del registro de Loggers.
- Los loggers se organizan en un espacio jerárquico de nombres.
  - Heredan propiedades de configuración y propagan mensajes.
- Los mensajes de error se asocian a niveles de importancia.
  - Permite filtrar mensajes por nivel.
- Permite enviar mensajes a consola, archivos, sockets.
- Permite ajustar el formato de los mensajes.
- Se puede extender (nuevos formatos, destinos, filtros).

```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");
    }
}
```

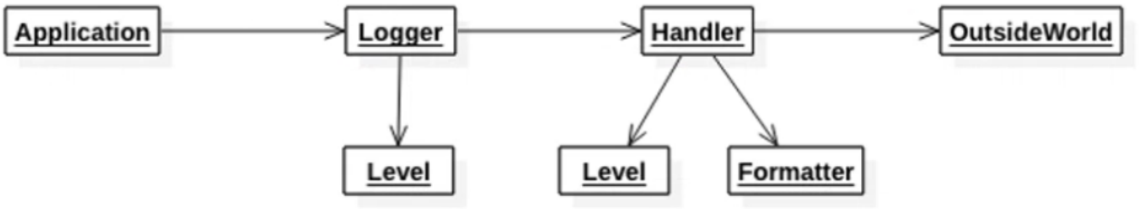
La configuración de los loggers suele ocurrir al inicial la aplicación, y se mantiene globalmente.

- A cada logger le puedo poner un nombre y elegir a dónde redirigir la salida del mismo.
- El Logger.getLogger ma da la instancia del logger. Si no existe lo crea y me lo manda.

En cualquier método, se puede recuperar y utilizar un logger. Lo que antes era el print.

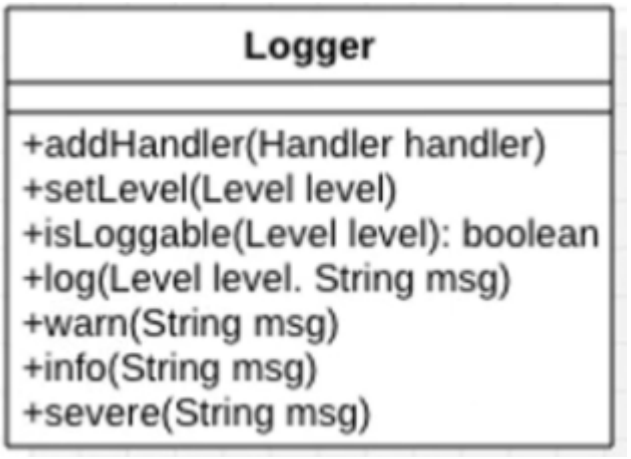
Partes básicas - Logger, Handler, Level, Formatter

- **Logger**: objeto al que le pedimos que emita un mensaje de log.
- **Handler**: encargado de enviar el mensaje a dónde corresponda. Puede haber varios por logger.
- **Level**: indica la importancia de un mensaje y es lo que mira un Logger y un Handler para ver si le interesa.
- **Formater**: determina cómo se "presentará" el mensaje. El formato del mensaje.



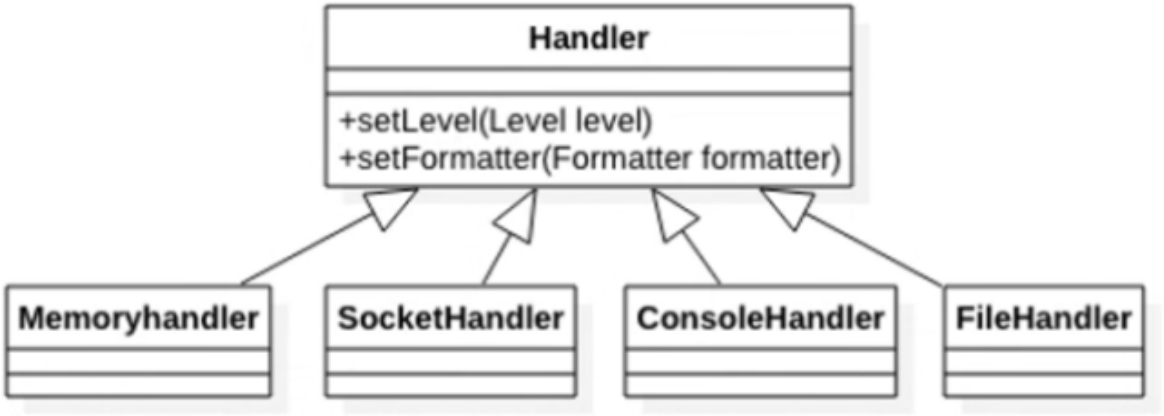
Logger

- Se pueden definir tantos como se necesiten.
  - Instancias de la clase Logger
  - Las obtengo con Logger.getLogger(String nombre)
- Cada uno con sus filtro y handler/s.
- Se organizan en un árbol (en base a su nombre)
  - Heredan configuración de su padre (handler y filters).
- Envío el mensaje log (Level, String) para que emita algún mensaje.
  - Alternativamente uso warn(), info(), severe(). Es hacer un log, pero ya se le asigna el nivel de importancia.



Handler

- Recibe los mensajes del Logger y determina cómo exportarlos.
- Instancias de MemoryHandler, **ConsoleHandler**, **FileHandler** o SocketHandler.
- Puede filtrar por nivel.
- Tiene un Formatter.

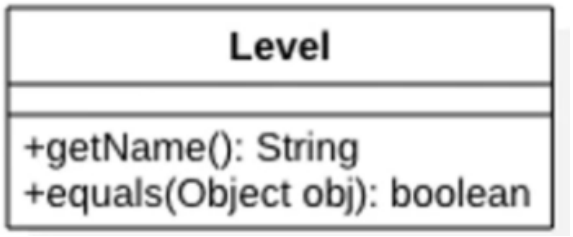


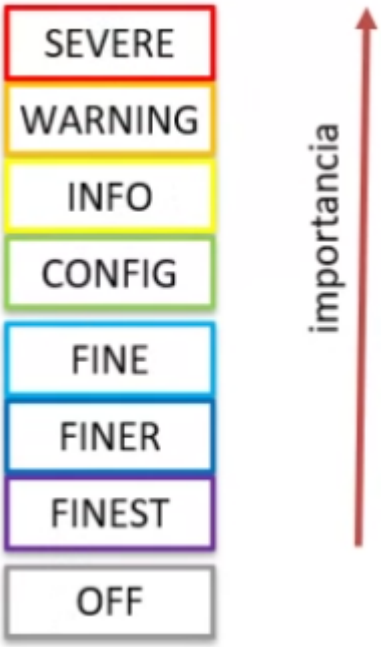
Level

- Representa la importancia de un mensaje.
- Cada vez que pido que se logee algo, debo indicar un nivel.
- Los Loggers y Handler comparan el nivel de cada mensaje con el suyo para decidir si les interesa o no. Si logger tiene una dificultad mayor, el mensaje

no pasa.

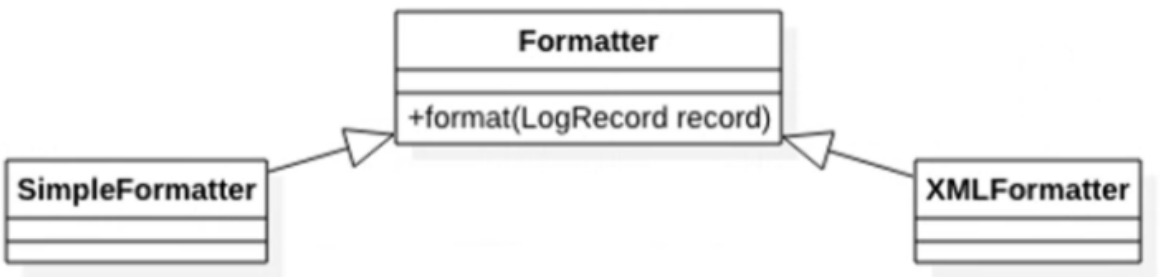
- Si interesa un nivel, también interesan los que son más importantes que ese.
- Hay niveles predefinidos, en variables estáticas de la clase Lever.





Formatter

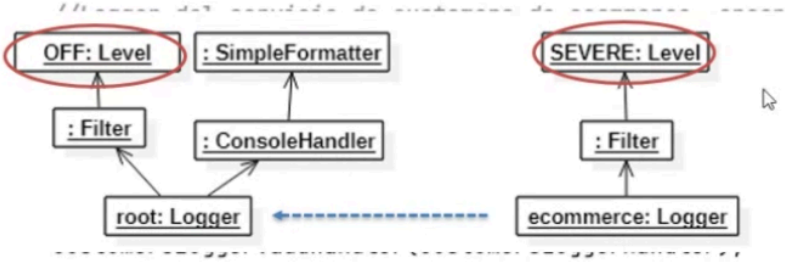
- Recibe un mensaje de log(un objeto) y lo transforma a texto.
- Son instancias de: SimpleFormatter o XMLFormatter.
- Cada handler tiene su formatter:
  - Los FileHandler tiene un XMLFormatter por defecto.
  - Los ConsoleHandler tienen un SimpleFormatter por defecto.



Ejemplo avanzado

```
//Loggers apagados por defecto
Logger.getLogger("").setLevel(Level.OFF);

//Loggers encendidos en nivel SEVERE para ecommerce
//Utilizará un ConsoleHandler y un SimpleFormatter
Logger  ecommerce = Logger.getLogger("ecommerce");
ecommerce .setLevel(Level.SEVERE);
```



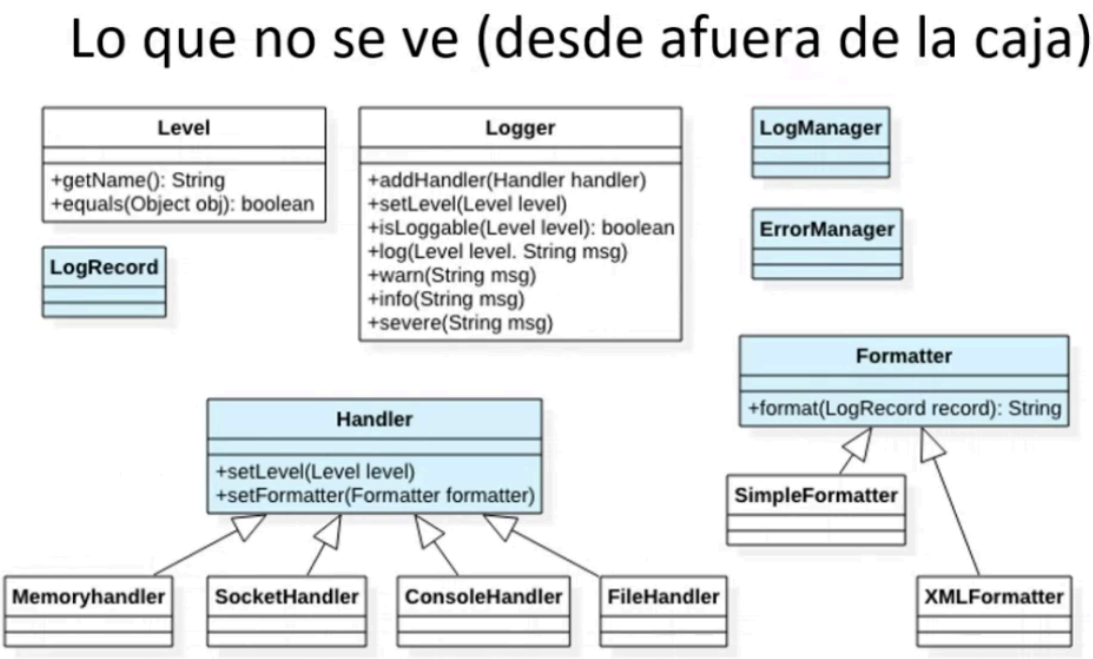
Todos los loggers heredan del logger con string " ". Si al root le modifico el handler para ir a disco, todos los loggers que no tengan handler van a ir a disco.


```
// Este logger hereda del raíz y no define nada propio, por lo tanto ignora el warning
Logger.getLogger("delivery").log(Level.WARNING, "Error in delivery");

// Este logger hereda de ecommerce por lo tanto ignora el warning
Logger.getLogger("ecommerce.products").log(Level.WARNING, "Stock inconsistency detected");

// A este logger le interesa el warning, que termina en un archivo con formato simple
Logger.getLogger("ecommerce.customers").log(Level.WARNING, "Stock inconsistency detected");
```

- El tema de la herencia de este framework forma parte de su frozenspot. Lo uso como quiera pero no puedo modificar la forma en la que se hace.



Extendiendo el framework (caja blanca)

Mirando adentro (caja blanca), puedo agregar nuevas clases de Formater, Handler y Filter.

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

Esto no es hacking, sino algo previsto por los diseñadores. Toda extensión (esperada) se va a adaptar a las futuras versiones.

Si yo modifico algo que no estaba previsto (modificar código fuente), probablemente lo que haga no funcione en un futuro, cuando salgan las nuevas versiones.

Nuevos Formatters

```java
public class ShoutingSimpleFormatter extends SimpleFormatter {
    @Override
    public String format(LogRecord record) {
        // SHOUTING WITH ALL UPPERCASE
        return super.format(record).toUpperCase();
    }
}
```

```java
public class JSONFormatter extends Formatter {
    @Override
    public String format(LogRecord record) {
        // Do whatever necesssary to represent record as
        // a JSON formatted string and return it
        return "...";
    }
}
```

- La primera muestra como extender a Formatter, creando uno que “grite” los Logs.
- En el caso de JSON hay que implementarlo en la práctica.