Frameworks

REÚSO

el reúso abarca todas aquellas técnicas, procesos y metodologías que tienen como objetivo reutilizar artefactos de software, creados en cualquiera de las etapas, para ser usados en nuevos desarrollos o en la construcción de nuevas versiones

reusamos:

- aplicaciones completas que adaptamos e integramos: Wordpress, GoogleApps, etc.
- componentes y servicios: Twitter API, Facebook API, Google Maps
- funciones y estructuras de datos: de colecciones, fechas, archivos, etc.
- diseños o estrategias de diseño: patrones, algoritmos o estructuras
- conceptos e ideas que funcionan: compartir en las redes sociales, ayuda en contexto

para qué

- para aumentar productividad, reducir costos, riesgo, incertidumbre y tiempos de entrega
- para aumentar calidad, porque el reúso aprovecha mejoras y correcciones

dificultades

- problemas de mantenimiento si no tenemos control de los componentes que reusamos
- algunos programadores prefieren hacer todo ellos mismos
- hacer software reusable es más difícil y costoso
- encontrar, aprender a usar y adaptar componentes reusables requiere esfuerzo adicional

algunas ejemplos de reúso

- patrones de diseño
- frameworks
- patrones arquitecturales
- software product line: métodos, herramientas y técnicas de ingeniería de software para crear una colección de sistemas de software similares
- librerías: es un conjunto de implementaciones funcionales, codificadas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se invoca
- program generators: un usuario puede especificar los pasos necesarios para el programa

LIBRERÍAS DE CLASES/ TOOLKITS

una librería de clases es un conjunto de clases que puedo usar independientemente pero todas tienen una relación conceptual

- resuelven problemas comunes de la mayoría de las aplicaciones como manejo de archivos, funciones aritméticas, colecciones o fechas
- cada clase resuelve un problema concreto, independientemente del contexto de uso
- nuestro código controla/ usa a los objetos de las librerías

librerías en Java
codec: implementan algoritmos de encoding/decoding
compress: clases para trabajar con archivos tar, zip, etc.
math: implementan componentes de matemática
java.util.collections: implementan estructuras de datos

FRAMEWORKS

un framework es una aplicación semi completa y reusable que puede ser especializada para producir aplicaciones a medida. O también se define como un conjunto de clases abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas

- las clases en el framework se relacionan de manera que resuelven la mayor parte del problema en cuestión
- el código del framework controla al nuestro

entonces, los frameworks son el resultado de refactorizar una aplicación, no solo mejora la calidad interna sino abstrae lo que es común en una familia de aplicaciones para así crear un esqueleto

FRAMEWORKS DE INFRAESTRUCTURA

brindan una infraestructura portable y eficiente sobre la cual construir una gran variedad de aplicaciones. Algunos enfoques son interfaces de usuario (desktop, web, móviles), seguridad, contenedores de aplicación, procesamiento de imágenes,

procesamiento de lenguaje, comunicaciones

FRAMEWORKS DE INTEGRACIÓN
se utilizan comúnmente para integrar componentes de aplicación distribuidos (la base de datos con la aplicación y ésta con su cliente liviano.). Están diseñados para aumentar la capacidad de los desarrolladores de modularizar, reusar y extender su infraestructura de software para que trabaje transparentemente en un ambiente distribuido

FRAMEWORKS DE APLICACIÓN
atacan dominios de aplicación amplios que son pilares fundamentales de las actividades de las empresas. Ejemplos de frameworks enterprise son aquellos en el dominio de los ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), gestión de documentos, cálculos financieros

Java
hibernate: framework de integración que resuelve el mapeo de objetos a bases de datos relacionales
jBPM: framework de aplicación (y suite de herramientas) para construir aplicaciones en las que se modelan, ejecutan, y monitorean procesos de negocios.
jUnit: framework de aplicación/infraestructura que resuelve la automatización de tests de unidad
IibGDX: framework de aplicación, para construir juegos en Java, multiplataforma
spring: familia de frameworks de infraestructura e integración, enfocando una amplia variedad de necesidades (Testing, Data, Web, etc.)

FROZENSPOT vs HOTSPOT

- los frozensports de un framework son aspectos que NO podemos cambiar, por lo tanto estas partes serán igual para todos los artefactos construidos con el framework.
- los hotspots son los *puntos de extensión/ ganchos* que nos permiten introducir variantes y así construir aplicaciones diferentes. Son las partes del artefacto a construir con el framework que puedo cambiar. Algunos *puedo* y otros *debo* instanciarlos

reconocer los hotspots nos permite diseñar mejor el framework y las aplicaciones que lo usan. Instanciar un hotspot puede requerir una combinación de acciones, de herencia y de composición

INVERSIÓN DE CONTROL
la inversión de control permite cambiar cosas que el desarrollador no tuvo en cuenta mediante la extensión del framework.
el código del framework controla el código que yo escribo
Gran parte del tiempo el control está en el código framework y cada tanto envía mensajes a objetos que yo tuve que programar para usar ese framework.
Cuando uno habla de artefactos re-utilizables debemos pensarlo con 2 gorros . El que desarrolla ese framework y quien lo usa.

- el que lo construye: tiene la complejidad de construir algo que sea re-usable y en cierta medida es incompleto y el usuario tiene que implementar clases o métodos para llegar a completarlo. Las decisiones de diseño me atan a futuro, no son tan difíciles de cambiar como las del que lo usa
- el que lo usa: hay ciertas decisiones que no se pueden cambiar acerca de como se construyen o cómo funciona, solo puedo cambiar o agregar variabilidad desde el lado del que diseñó el framework dejó previsto

CAJA BLANCA vs. CAJA NEGRA
los puntos de extensión pueden implementarse en base a herencia o composición.

- HERENCIA: los framework que implementan los hotspots mediante herencia se conocen como *white box*, por lo cual se usan variables y métodos heredados. Se pueden extender, implementar y redefinir métodos lo cual lo lleva a la **inversión de control.**
  Los ganchos están en las clases abstractas y concretas.
  Se pasa el control de los mensajes son *self* y no es necesario pasar el estado como parámetro
  Es importante documentar ya que el framework espera determinadas cosas del código del usuario. Por esto mismo no se puede cambiar el diseño como si nada
  Para todo esto usamos patrones como *template method* ya que asegura que cumplamos todos los pasos. Programamos por diferencia implementando los *hooks*, usando herencia. Tenemos acceso a las variables de instancia y métodos que

heredamos pero nuestra jerarquía explota si hay muchas combinaciones y NO podemos cambiar el comportamiento en el tiempo de ejecución

- COMPOSICIÓN: los framework que implementan los hotspots mediante composición se conocen como *black box*, para usarlos se necesita instanciarlos y configurarlos
  El paso de control es *mediante* callbacks por eso es necesario pasar y actualizar los estados necesarios en estas llamadas
  Solo se conocen algunas clases del framework y sus mensajes, por lo cual no puedo cambiarlo ni extenderlo. Como usuario se sabe poco por lo cual se puede cambiar el framework sin preocuparse demasiado.
  No se necesita mucha explicación o documentación y se hereda donde sea necesario. Los framework terminan evolucionando cada vez más en uno de caja negra, es más difícil de desarrollar pero más fácil de usar
  En este caso tal vez implementamos con algún strategy o state. No tenemos acceso a las variables de instancia ni métodos privados, dependemos de las opciones provistas pero PODEMOS cambiar el comportamiento en run-time.

la herencia es el mecanismo principal para separar lo que es constante de lo que varía. El método plantilla es parte del frozenspots mientras que las operaciones abstractas (o los ganchos) dan lugar a la implementación de los hotspots

si uso composición, un objeto implementa la plantilla y delega la implementación de los ganchos en sus partes.
si uso herencia, la plantilla se implementa en una clase abstracta y los ganchos en sus subclases.

HOOKS
son funciones que permiten enganchar nuestros componentes funcionales a características propias de un componente de clase

COMO USUARIOS DEL FRAMEWORK

HERENCIA	**COMPOSICIÓN**
• implemento, extiendo y redefino métodos • uso variables y métodos heredados • podría cambiar cosas que el desarrollador no tuvo en cuenta • puedo extender el framework • debo aprender qué heredo y qué puedo hacer con ello • no puedo heredar comportamiento de otro lado	• instancio y configuro • conecto a mi código con callbacks • solo conozco algunas clases del framework y sus mensajes • no puedo cambiar o extender el framework • solo tengo acceso a los objetos que recibo de los callbacks

COMO DESARROLLADORES DEL FRAMEWORK

HERENCIA	**COMPOSICIÓN**
• dejo ganchos en clases del framework • paso el control con mensajes a self • no necesito pensar todos los hotspots y todos sus casos • no necesito pasar estado como parámetros • debo documentar claramente qué se puede tocar o no • no puedo cambiar el diseño sin preocuparme por los usuarios	• dejo objetos configurables para ajustar el comportamiento • paso el control con callbacks • debo pensar todos los hotspots y sus casos • debo pasar/ actualizar todo el estado necesario en los callbacks • no sé nada del código del usuario • no necesito explicar cómo funciona • puedo cambiar mi diseño sin preocuparme

en resumen, un framework tendrá hotspots que se instancian con herencia y otros que se instancian por composición, por lo general arrancan dependiendo mucho de herencia para ir evolucionando a composición, es más fácil desarrollarlos si son caja blanca y usarlos si son caja negra, es más desafiante desarrollarlos si son caja negra y usarlos si son caja blanca

LOGGING

los log se usa para referirse a la grabación secuencial en un archivo o en una base de datos de todos los acontecimientos que afectan a un proceso particular
los agregamos para entender lo que pasa con ella, por ejemplo para reportes de eventos, errores y excepciones, pasos críticos o inicio y fin de operaciones
resultan útiles para los desarrolladores, administradores y usuarios

no reemplaza al testing
podríamos usar System.out.println pero es mejor incluir los frameworks de logging, los cuales permiten estandarizar la practica, activar/ desactivar logs, enviar reportes en distintos formatos y ocultar detalles de implementación

JAVA LOGGING FRAMEWORK
es un framework incluido en el SDK

- se organizan en un espacio jerárquico de nombres
- los mensajes de error se asocian a niveles/ importancia
- permite enviar mensajes a consola, archivos y sockets
- es entendible

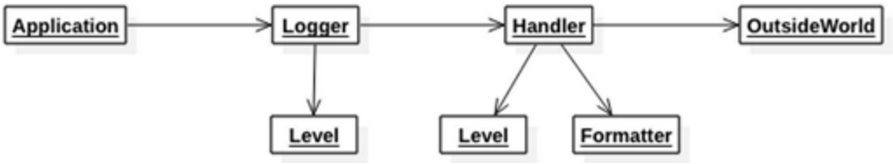
```java
public class Sandbox {
    public static void main(String[] args) throws IOException {
        Logger.getLogger("app.main").addHandler(new FileHandler("log.txt"));
        Logger.getLogger("app.main").log(Level.INFO, "App iniciada");
        try {
            // Acá que hace algo que "podría" 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");
    }
}
```

PARTES VISIBLES

- logger: objeto al que le pedimos que emita un mensaje de log, podemos definir tantos como necesitemos, son instancias de la clase Logger
  - las obtengo con Logger.getLogger(String name)
  - cada uno con su filtro y handler/s
  - se organizan en un árbol a base de sus nombres y heredan configuración de su padre
  - envío el mensaje log(Level, String) para emitir algún mensaje
- handler: encargado de enviar el mensaje a donde corresponda. Recibe los mensajes del Logger y determina cómo "explotarlos", son instancias de MemoryHandler, ConsoleHandler, FileHandler o SocketHandler, puede filtrar por nivel y tiene un formatter
- level: indica la importancia de un mensaje y es lo que mira un Logger y un Handler para ver si le interesa. Representa la importancia de un mensaje, cada vez que pido que se loggee algo, debo indicar un nivel. Los loggers y Handler miran el nivel de un mensaje para decidir si les importa o no
  - los niveles posibles son: Level.SEVERE, Level.WARNING, Level.INFO, Level.CONFIG, Level.FINE, Level.FINER, Level.FINEST (el menos importante), y Level.OFF (nada)
  - si te interesa un nivel, también te interesan los que son más importantes que ese
- formater: determina cómo se presentará el mensaje. Recibe el mensaje de log 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

PARTES NO VISIBLES

- LogRecord: objeto que representa un mensaje
- LogManager: es el objeto que matiene el árbol de loggers; hay un solo LogManager (global)
- Filter: Logger y Handler tienen un Filter para determinar si algo les interesa o no (el Filter conoce al Lever)
- ErrorManager: los Logger pueden tener un ErrorManager para lidiar con errores durante el logging
- Clases abstractas: Handler y Formatter

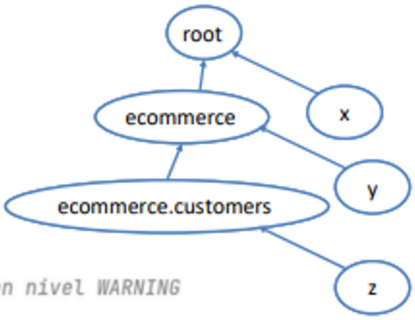


ejemplo

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

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

//Logger del servicio de customers de ecommerce, encendido en nivel WARNING
//con destino un archivo, en formato simple texto
Logger customersLogger = Logger.getLogger("ecommerce.customers");
customersLogger.setLevel(Level.WARNING);
FileHandler customersLoggerHandler = new FileHandler("ecommerce-customers.log");
customersLoggerHandler.setFormatter(new SimpleFormatter());
customersLogger.addHandler(customersLoggerHandler);
```



- getLogger() es un método estático presente en la clase Logger que crea un registrador si no está presente en el sistema con el nombre dado, en este caso "ecommerce"
- FileHandler envía mensajes al archivo que le indiquemos
- SimpleFormatter() para textos de plano simple y no en formato XML
- tanto el Logger como el Hander tienen un level para decidir si están interesados en un determinado logRecord.
- handler puede usar un Formatter para localizar y formar un mensaje
- los niveles asignan la importancia y urgencia de un mensaje de log
- tipos de handler:
  - StreamHandler: un simple handler para escribir registros en un OutputStream
  - ConsoleHandler: handler para escribir registros en System.err
  - FileHandler: handler que escribe registros en un archivo
  - SocketHander: escribe registros en TCP ports
  - MemoryHander: los escribe en memoria
- formatos:
  - SimpleFormatter: escribe resúmenes "entendibles" para las personas
  - XMLFormatter: escribe información estructurada en XML

FROZENSPOTS:

- diseño/ propuesta sobre cómo integrar logging
- estrategia general, siempre busco un logger para configurarlo o pedirle que emita un mensaje
- tengo logger,handler, formatter, filter y level
- cómo se organizan los loggers (árbol)
- las configuraciones heredadas
- qué pasa cuando le digo log() a un logger

HOTSPOTS:

- cuántos y cuáles logreras uso
- qué logger "hereda" cual
- qué le interesa a cada logger
- qué handlers se asocian a cada logger
- qué formatter tiene cada handler y qué le interesa

RESUMEN

java.util.logging es un framework que propone una forma de integrar logging en nuestros programas, lo uso mayormente como una caja negra y toma el control cuando le indicamos y lo devuelve cuando termina. Puedo extender el framework heredando en los puntos de extensión previstos