Segnalazione di errori in log4j

Come “rompere lo schema” e lanciare eccezioni in caso di fallimento

Utilizzo di *Framework*

- *API* ben specificate
  - Implementabili
  - Non modificabili
- Flusso dell'applicazione
  - Spesso configurabile
  - Non alterabile a piacimento

Robustezza

A volte I *Framework* “acchiappano” le eccezioni

- *Logging* non riuscito
- Messaggi ridondati non pervenuti
- Errori non preoccupanti

Il concetto di non preoccupante non sempre e’ universalmente accettato.

Generalizzando

I *Framework* potrebbero mascherare la comparsa di certi eventi ritenuti (dal *Framework*) *non importanti*.

Un cliente del *Framework* potrebbe essere interessato alla notifica dell'evento (a torto o a ragione).

Estensioni

- E’ possibile estendere parti dei *Framework*
- Molti *Framework* dispongono di un package *.spi* (*Service Provider Interface*)

Tuttavia non si possono alterare le API o il flusso del Framework!

Iniettare componenti

Spesso le componenti spi vengono iniettate in modo programmatico (esplicito) o tramite file di configurazione (implicito, *dependency injection*), di norma sono disponibili entrambi i metodi.

Iniettare il nostro componente

Si deve quindi provvedere ad iniettare il giusto componente nel *Framework*.

Tale componente deve essere

- Non invasivo
- *Thread safe*
- Corretto
- Ben identificabile

Ben identificabile?

Essendo il nostro componente un *alieno* per il *Framework*, questo difficilmente ci permetterà di recuperare la giusta istanza del nostro componente.

I *Framework* consentono l'iniezione di istanze diverse della solita classe.

Design Patterns

- Soluzioni riusabili a problemi noti
- Oramai non sono più “roba accademica”
- Hanno nomi noti e codici riconoscibili
- Uno di questi *pattern* può esserci d’aiuto?

Singleton Pattern

```
public class SimpleSingleton {
        private static final SimpleSingleton _instance =
                new SimpleSingleton();
        protected SimpleSingleton(){};
        public static SimpleSingleton getInstance() {
                return _instance;
        }

}
```

Nota

Lo scopo di questo documento non è quello di fornire buone pratiche sulla programmazione Java. Tuttavia**Singleton** come il precedente sono “da dilettanti”. Si consiglia di far istanziare il **Singleton** ad un’altra classe che funge da **Factory** in modo da sfruttare a pieno il *Lazy Class Loading* di Java. (Attenzione! Non confondete il *Lazy Class Loading* con la *Lazy Instantiation*)

Singleton non sempre iniettabile

Il **Singleton** non è sempre iniettabile in quanto privo di costruttore.

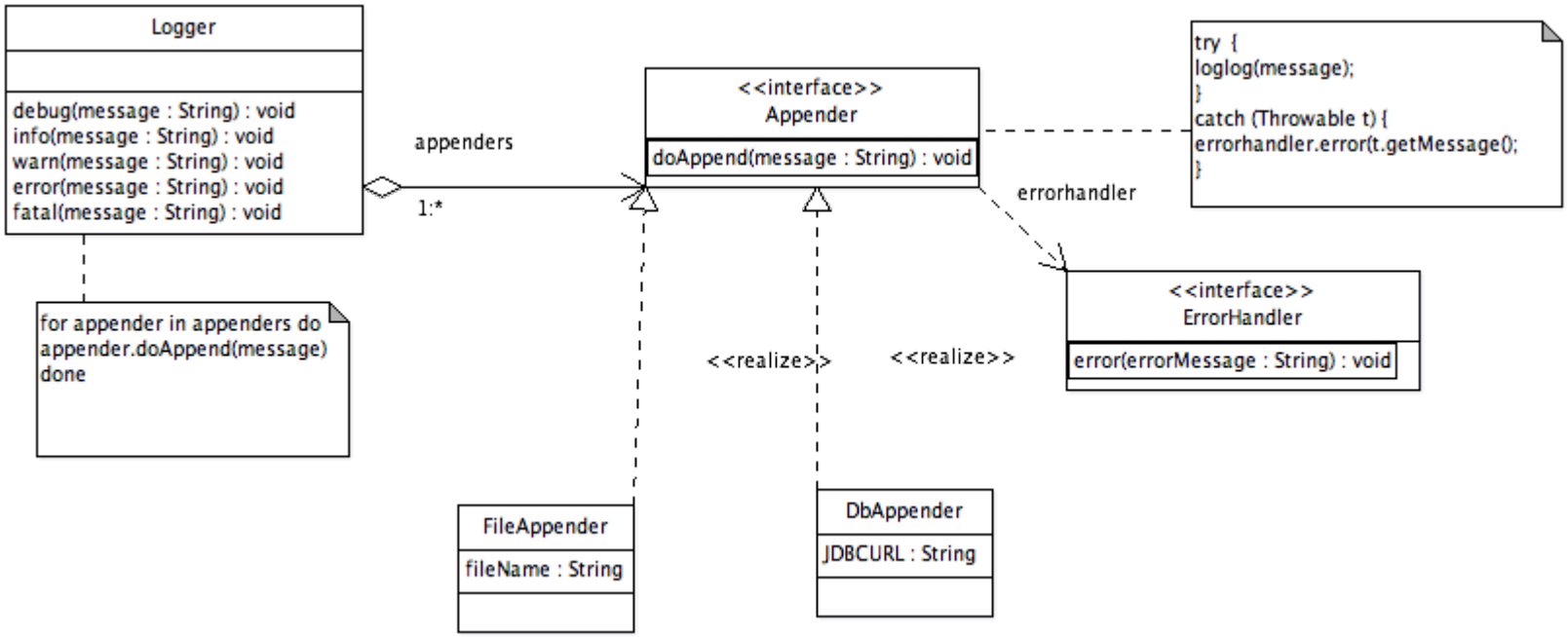
Il metodo di configurazione utilizzato dal *Framework* potrebbe non disporre di un modo per invocare la *getInstance()*. (Ad esempio log4j ma non Spring)

Esiste un altro Design Pattern che ci può aiutare?

Delegation

```
public class DelegateSingleton {
        private static final SimpleSingleton _delegate =
                SimpleSingleton.getInstance();
        public void DelegateSingleton() {};
        public void doSOmething(){
                return _delegate.doSomething();
        }
.............
```

Sketch UML delle classi di log4j



Classi ed interfacce

- **Logger**: riceve messaggi da parte del client utente (classe Java). Delega la scrittura dei messaggi ad uno o più **Appender**.
- **Appender**: riceve messaggi da vari **Logger** e li "appende" su opportune fonti dati. In caso di fallimento delega la gestione dell'errore ad un opportuno **ErrorHandler**.
- **ErrorHandler**: gestisce i fallimenti nelle scritture dei messaggi di log.

Numero di istanze

- **Logger**: dipende dal contesto
- **Appender**: una istanza per ogni appender nominale.
- **ErrorHandler**: una istanza per ogni errorhandler nominale

Nota: gli **Appender** si sincronizzano sull'argomento implicito del metodo di scrittura dei log. Se ogni **Appender** scrive su di un file diverso non si hanno problemi di accesso concorrente.

Istanza singola?

SI! Come nelle Servlet e simili.

L'istanza singola garantisce di norma migliore performance e migliore gestione della memoria.

Tuttavia non è possibile iniettarci dentro qualcosa e sperare di recuperarla senza problemi.

Soluzione proposta

- Nessun **Appender** *Custom*
- Un **ErrorHandler Singleton**
- Un **ErrorHandler** non **Singleton** che delega tutte le chiamate a quello **Singleton**
- Metodo aggiuntivo che permette di recuperare dal nostro **ErrorHandler** informazioni sullo stato della “loggata”

Doppio ErrorHandler?

- Utili per configurazione programmatica o implicita
- Si utilizza l'uno o l'altro a seconda del tipo di configurazione o in base al gusto personale
- L'importante è che tutto venga delegato al **Singleton**

Ed i Thread?

- *Race Condition*
- **Singleton** -> mutua esclusione
- 1 istanza per appender -> no mutua esclusione

E allora?

ThreadLocal

Java.lang.ThreadLocal

E' una classe Java che modella variabili "locali" al Thread

Un Thread esegue una sola operazione alla volta

Le **ThreadLocal** di un Thread possono essere viste solo dal Thread che le possiede

Mettiamo tutto assieme

L'ErrorHandler Singleton contiene una variabile **ThreadLocal** contenente lo "stato di errore". (supponiamo che sia una stringa che vale null in caso di successo)

Dopo una "loggata" viene letta tale variabile dal client di log4j.

Se la variabile contiene un errore allora viene lanciata un'eccezione.

E’ automatizzabile?

SI

- Creando wrapper della classe Logger
- Estendendo la classe Logger

Esempio di ErrorHandler

```
public class CustonErrorHandler {
private static final ErrorSingleton _instance = new ErrorSingleton();
public static ErrorSingleton getInstance() {
        return _instance;
    }
public static class ErrorSingleton implements ErrorHandler {
        protected ErrorSingleton() {}
        ThreadLocal<String> errore = new ThreadLocal<String>() ;
        public void error(String arg0) {
            errore.set(arg0);
        }
        public String getError() {
            return errore.get();
        }
……….
```

Come si utilizza?

```
CustonErrorHandler.ErrorSingleton eh =
    CustonErrorHandler.getInstance();
logger.debug("CIAOOO");
String errore = eh.getError();
if (errore != null){
    throw new RuntimeException(errore) ;
}
.....
```