Obiective

Scopul acestui laborator este experimentarea unui mecanism **complementar** testarii, pentru detectarea bug-urilor si validarea unui program.

Subiecte atinse:

- *logging*
- pachetul log4j

Logging

Logarea reprezinta inregistrarea de informatii din timpul rularii programelor: starea sistemului la un moment dat, aparitia unui eveniment etc. Cu totii am folosit o forma primitiva de logare: instructiunea print pentru a afisa valoarea unei variabile in diferite puncte din program.

Logarea este folosita cu dublu **scop**:

- ***debugging***: urmarirea fluxurilor de control si date ale programului
- ***auditing***: desprinderea unor informatii utile din evolutia sistemului, ca, de exemplu, utilizatorul care a executat o anumita actiune

Experienta demonstreaza ca, in numeroase cazuri, logarea se doveste mai utila decat utilizarea unor tool-uri de ***debugging***, deoarece se poate accesa informatia care intereseaza, in momentele relevante din executia programului, evitand activitatile mai complexe, specifice procesului de *debugging*, cum este avansul instructiune cu instructiune. Logarea se dovedeste si mai utila in situatia in care *debugging*-ul este foarte anevoios; este cazul aplicatiilor *multithreaded* sau distribuite. **Mesajele** de logare pot fi afisate la consola, scrise intr-un fisier, sau chiar trimise pe retea.

Desigur, exista si **dezavantaje** ale acestui procedeu:

- **diminuarea vitezei de rulare**
- **dificultatea urmaririi output-ului**, daca este prea detaliat.

log4j

log4j reprezinta un mecanism de logare pentru Java. Il puteti obtine de la http://logging.apache.org/log4j/1.2/download.html. Gasiti o descriere introductiva la adresa http://logging.apache.org/log4j/1.2/manual.html. Pachetul exista si in variante corespunzatoare altor limbaje: log4cxx, log4php etc.

Elemente de baza

log4j ofera posibilitatea de a:

- defini ***logger*-e**, obiecte ce realizeaza logarea. Pot fi **ierarhizate**, pe baza numelui acestora

- asocia **prioritati** mesajelor de logare si a stabili un **prag** al acestor prioritati, astfel incat sa se genereze doar mesajele a caror prioritate depasesc pragul
- asocia unul sau mai multe ***appender*-e** cu un *logger*, acestea stabilind **destinatia** output-ului: consola, fisier, retea etc.
- stabili **formatul** mesajelor de logare, folosind ***layout*-uri**
- defini **fisiere de configurare**, ce manipuleaza comportamentul de logare la rulare, fara modificarea codului sursa

Iata un prim exemplu de program:

```
import org.apache.log4j.*;


public class Test {
   static Logger logger = Logger.getLogger(Test.class); //sau Logger.getLogger("Test");


   public static void main(String[] args) {
      BasicConfigurator.configure();
      logger.info("Hello");
   }
}
```

Se observa:

- directiva import. De asemenea, trebuie sa aduagati fisierul log4j-1.2.15.jar la CLASSPATH
- clasa Logger si metoda statica getLogger. Aceasta primeste ca parametru **numele** *logger*-ului, care, cel mai adesea, este numele clasei care il contine
- apelul metodei info, care logheaza mesajul dat ca parametru, semnaland **prioritatea** INFO. Alte prioritati sunt: DEBUG < INFO < WARN < ERROR < FATAL. Pe langa acestea mai exista ALL si OFF, cel din urma folosindu-se cand se doreste dezactivarea logarii pentru mesajele asociate unui anumit obiect *logger*
- apelul BasicConfigurator.configure(), care ii asociaza *logger*-ului un *appender* de consola. Este suficienta o **unica** invocare pentru intreaga aplicatie. Aceasta are loc, de obicei, in metoda main.

Output-ul default este:

```
0 [main] INFO Test - Hello
```

avand semnificatia:

durata_de_la_inceputul_executiei [thread] prioritate nume - mesaj

Metoda getLogger reflecta 2 **design patterns**:

- ***factory***: produce obiecte pe baza parametrului
- ***singleton***: pentru acelasi nume, returneaza intotdeauna **aceeasi referinta**

Prioritati (log levels)

Prioritatile, mentionate mai sus, sunt asociate, de obicei, urmatoarelor **situatii**:

1. FATAL: esec ce conduce la **terminarea** aplicatiei. Exemplu: imposibilitatea de a incarca un modul al aplicatiei.
2. ERROR: eroare ce permite **continuarea** aplicatiei, fiind asociata, de obicei, unei exceptii Java. Exemplu: esecul conectarii la baza de date. Legatura poate fi restabilita ulterior.
3. WARN: probleme **minore**, externe aplicatiei. Exemplu: furnizarea unor date incorecte de catre utilizator
4. INFO: eveniment din fluxul **normal** de executie a aplicatiei. Exemplu: citirea parametrilor de configurare
5. DEBUG: informatie de o granularitate foarte **fina**. Exemplu: valoarea unei variabile la un anumit moment

Pragul de prioritate (*log level*) **inhiba** generarea mesajelor cu prioritate **inferioara**. Daca, spre exemplu, am adauga secventa:

```
logger.setLevel(Level.FATAL);
logger.info("Hello");
```

mesajul Hello nu ar mai fi afisat, deoarece nivelul INFO este inferior celui FATAL.

Se poate construi o **ierarhie de *logger*-e**, botezandu-le cu nume separate prin caracterul ".", ca in cazul pachetelor si claselor Java. De exemplu, *logger*-ul idp.gui este parinte pentru *logger*-ul idp.gui.Gui. Exista un *logger* parinte, universal, ce poate fi obtinut prin metoda getRootLogger. Relatia ierarhica se refera la **mostenirea**:

- tuturor ***appender*-elor** de la toti stramosii
- **pragului de prioritate** de la parintele direct

Appender-e si layout-uri

Un ***appender*** reprezinta o **destinatie** posibila a mesajului de logare, iar un obiect *logger* poate poseda **oricate** *appender*-e. Exemple:

- ConsoleAppender: pentru afisarea la consola

- FileAppender: pentru scrierea in fisier
- RollingFileAppender: permite generarea unor fisiere de log de o **dimensiune maxima**, urmand ca, la depasirea acesteia, sa se comute la alt fisier de log. De asemenea, permite precizarea **numarului maxim** de astfel de fisiere, ce se doresc pastrate.
- JDBCAppender: pentru logare intr-o baza de date
- SocketAppender: pentru trimiterea mesajului pe retea.

Pragul de logare poate fi stabilit si la nivel de *appender*, folosind parametrul Threshold.

Un ***layout*** este **asociat** unui *appender* si determina **formatarea** mesajului respectiv. Exemple uzuale:

- SimpleLayout: sirul logat contine doar mesajul original
- PatternLayout: permite imbogatirea mesajului cu informatie suplimentara, precum numele fisierului, al metodei curente, al firului de executie etc. Formatarea se realizeaza prin indicatori asemanatori celor folositi de printf (vezi fisierul de configurare din sectiunea urmatoare).

Fisiere de configurare

Prezentam, in continuare, un fisier de configurare (log4j.properties - nume consacrat), aceasta fiind modalitatea recomandata, pentru a **evita modificarea** codului sursa cand se doreste schimbarea comportamentului:

log4j.properties

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger = DEBUG, A1


# A1 is set to be a ConsoleAppender.
log4j.appender.A1 = org.apache.log4j.ConsoleAppender


# A1 uses PatternLayout.
log4j.appender.A1.layout = org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern = %-4r [%t] %-5p %c %x - %m%n
```

Comportamentul este acelasi ca in cazul BasicConfigurator.configure(), apel care acum trebuie inlocuit cu PropertyConfigurator.configure(„log4j.properties“). Pentru a intelege semnificatia specificatorilor de mai sus, cititi aici.

Diagnostic Contexts

In cazul in care aplicatia ruleaza pe **mai multe fire**, devine utila evidentierea mesajelor de logare referitoare la aceleasi prelucrari. Simpla tiparire a numelui firului de executie, in dreptul mesajului, este insuficienta in cazul in care se intrebuinteaza un mecanism de reciclare a firelor. Ar fi utila posibilitatea definirii unor **identificatori** ai fiecarui fir, care sa insoteasca fiecare mesaj de logare.

In sprijinul acestei probleme vine clasa MDC (*mapped diagnostic context*), ce ofera functionalitatea unui **dictionar** (*map*), cu precizarea ca asocierile cheie-valoare se mentin la nivel de **fir**. De remarcat ca metodele clasei sunt **statice**:

```
MDC.put("name", "Deirdre"); // per thread

// layout-urile pot fi comandate sa tipareasca aceasta informatie, prin indicatori ca %X{name}

logger.info("Hello");

MDC.remove("name");
```

Un output posibil este:

```
Deirdre: Hello
```

Exercitii

Prezentare

Urmariti prezentarea pentru intelegerea conceptelor utilizate in laborator (http://elf.cs.pub.ro/idp/_media/laboratoare/l6/prezentare_lab6.pdf).

Enunturi

Folositi arhiva de la http://elf.cs.pub.ro/idp/_media/laboratoare/l6/idp_lab_6_skel.zip ca si schelet pentru laborator. Aici gasiti solutia laboratorului trecut, mai putin partea de *unit testing*.

1. (**2p**) Adaugati instructiuni de *logging*, cu **diferite** prioritati, in mai multe puncte ale programului (inclusiv in clasa ListWorker), pentru a putea urmari un lant **complet** de apeluri, de la interfata grafica pana la executarea actiunii (GUI - mediator - *state manager* etc.).
    - Botezati *logger*-ii cu numele claselor in care sunt definiti.
    - Afisati mai intai la consola (BasicConfigurator).

2. (**2p**) Creati un fisier log4j.properties, la **acelasi nivel** cu fiserele .class, cu continutul din textul laboratorului. Plasand fisierul in aceasta locatie, va fi citit automat dupa **inlaturarea apelului** BasicConfigurator.configure(). Ce reprezinta rootLogger din fisier? Schimbati *log level*-ul (prioritatea) si rulati aplicatia.
3. (**2p**) Modificati fisierul de configurare, prin adaugarea un ***file appender*** (manual log4j). Rulati.
    - Adaugati un ***pattern layout*** pentru noul *appender*, asemanator cu *layout*-ul pentru ConsoleAppender, din fisier, si tipariti numele fisierului, numele metodei si linia din fisier (cititi despre PatternLayout). Studiati fisierul de log.
4. (**2p**) Adaugati un ***rolling file appender***, ce permite generarea unor fisiere de log de o **dimensiune maxima**, urmand ca, la depasirea acesteia, sa se comute la alt fisier de log. De asemenea, permite precizarea **numarului maxim** de astfel de fisiere, ce se doresc pastrate.
    - Stabiliti, doar pentru acest *appender*, un prag de logare mai mare. **Hint**: parametrul Threshold.
5. (**2p**) Folositi un *mapped diagnostic context* pentru **gruparea** mesajelor de logare, provenite de la diferiti SwingWorker-i.

Resurse utile

- Pagina oficiala log4j (http://logging.apache.org/log4j/)
- Download log4j (http://logging.apache.org/log4j/1.2/download.html)
- Manual log4j (http://logging.apache.org/log4j/1.2/manual.html)