PRÀCTICA 3 – Documentació:

Introducció:

Aniré comentant diferents coses que he anat aprenent per Internet sobre la teoria que m'han semblat interessants i que no estaven en el contingut en general que s'ha proposat a teoria.

Índex:

Swing

Què és?

Swing és un joc d'eines per a Java. És part de Sun Microsystems Java Foundation Classes (JFC), que és un API per a proporcionar una interfície gràfica d'usuari (GUI) per a programes de Java.

Swing va ser desenvolupat per a proporcionar un conjunt més sofisticat dels components de les GUI en comparació amb l'anterior Window Toolkit Resum. Inclou widgets per a la interfície gràfica d'usuari com ara caixes de text, botons, desplegables i taules.

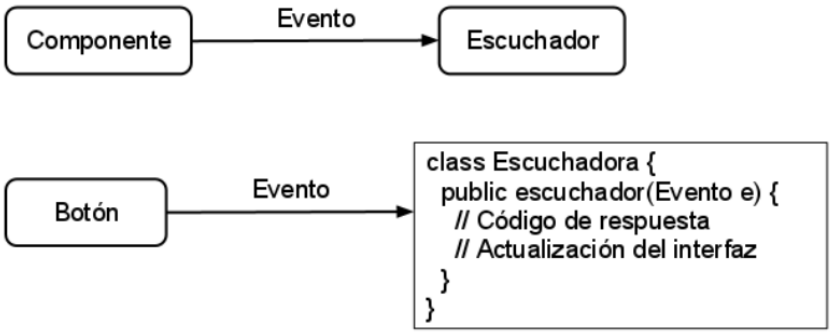
Algunes dels seus avantatges són:

- El disseny en Java pur posseeix menys limitacions de plataforma.
- El desenvolupament de components Swing és més actiu.
- Els components de Swing suporten més característiques.

IMPORTANT:
El model de programació de la resposta a esdeveniments a Swing, segueix el patró de disseny Observador / Observable. Els components Swing són Observables, quan l'usuari interacciona sobre ells, tots els Observadors són informats.

Swing utilitza l'aproximació push (explicada en la pràctica 1), en el moment d'informar els Observadors, se'ls envia una descripció del que ha passat. Per a completar el patró, és possible registrar i eliminar escoltadors als components.

El cicle d'interacció amb els components Swing:



La pràctica totalitat dels components Swing generen esdeveniments quan l'usuari interacciona amb ells. Els esdeveniments són instàncies de classes definides en el paquet estàndard de Java. Els contenidors també generen esdeveniments, però no ens ocupem d'ells. Si volem escoltar un determinat tipus d'esdeveniments, hem de definir una classe especialitzada en això.

La descripció per passos:

- Un component genera un esdeveniment (objecte)
- El component envia l'esdeveniment a tots els escoltadors registrats.
- Un escoltador és un objecte d'una classe que implementa una interfície concret.

Com podem conèixer els esdeveniments que llancen els diferents components?

Com pots veure, diferents components poden generar el mateix tipus d'esdeveniment. Fixa't en el nomenat, si el nom de la interfície és xxxListener l'esdeveniment que escolta és xxxEvent, i el mètode de registre és addxxxListener (xxxListener):
ActionListener → ActionEvent → addActionListener (ActionListener escoltador).

Procediment per a treballar amb Swing

En general, en treballar amb Swing el procediment és:

- Decideix quin component, o contenidor, vols utilitzar en la interfície gràfica.
- Busca els esdeveniments i escoltadors que se li poden registrar.
- D'entre tots, tria el, o els, que t'interessi.
- Escriu una classe que implementi els escoltadors que has triat, recorda, això implica que la teva classe implementi algunes interface.
- El teu codi de resposta el escriuràs en definir els mètodes que declara la interfície.
- Obre un instància de la teva classe escoltadora sobre el component.

T'has quedat amb ganes d'aprendre'n més? Tot això és un breu resum de:
http://www3.uji.es/~belfern/Docencia/Presentaciones/ProgramacionAvanzada/Tema3/swing2.html#32

Xat (Client – Servidor)

L'objectiu és crear un xat amb model client – servidor amb un thread per a cadascun. Aquest servidor serà capaç de gestionar múltiples clients, i l'estructura serà diferent que el que he creat a la pràctica 2 per a fer-lo força més simple, així poder centrar-me en la part gràfica de Swing que és la nova part apresa.

En diferència amb la pràctica anterior (n'hi ha moltes de diferències que aniré comentant en el document), sobretot vull destacar la quantitat de Threads que hi haurà.

Servidor

Per a resumir, el servidor estarà constantment escoltant peticions de clients noves i per a cada petició respondrà amb un worker (socket particular pel client), que mantindrà la connexió amb aquest. En general, el **WORKER** hauria de tenir aquesta pinta:

```
// Implementa Runnable per a poder ser llançada en un fil apart
public class Worker implements Runnable {
  // En el constructor es reben i es guarden els paràmetres que siguin necessaris
  // En aquest cas una llista amb tota la conversació y el socket que ha d'atendre.
  public Worker(DefaultListModel conversacio, Socket socket) {      // [...]
  }
  public void run ()
  {
    while (true)
    {
      // Codi per a atendre al client.
    }
  }
}
```

Una vegada tenim el worker preparat, el servidor simplement ha de ficar-se en un bucle infinit acceptant connexions i creant workers. Hauria de ser una cosa semblant a això:

```
public class ServidorChat
{
  // Per a guardar tota la conversació:
  private DefaultListModel conversacio = new DefaultListModel();
  public ServidorChat()
  {
    // Es crea el socket servidor, és el encarregat d'escoltar tota la estona:
    ServerSocket socketServidor = new ServerSocket(2900);
    // Bucle infinit
    while (true)
    {
      // S'espera y accepta un nou client:
      Socket client = socketServidor.accept();
      // S'instancia una classe per a atendre al client i es llança en un fil apart:
      Runnable nouClient = new Worker(conversacio, client);
      Thread fil = new Thread(nouClient);
      fil.start();
    }
  }
}
```

Sobre el codi anterior, s'observa una de les tasques a usar en la nostra pràctica, que consisteix en la classe **DefaultListModel**, que usa un patró observador, això vol dir que ens és possible ser assignats a una llista de tal manera que quan canviïn coses, nosaltres podrem ser notificats i actuar.

Cada classe Worker rep aquesta llista i en la pràctica això voldrà dir que quan arribi un text del client, aquest ficarà el contingut a la llista (al final). Com que tots els workers estaran subscrits a aquesta llista, inclòs el que ha enviat el missatge, tots seran assabentats del event i actuaran en conseqüència, enviant el text al seu client associat. Amb aquesta informació, he entès que cal afegir al **worker** que teníem abans amb:

```
// Implementa també ListDataListener per a poder subscriure's als canvis de DefaultListModel
public class Worker implements Runnable, ListDataListener {
  public Worker(DefaultListModel charla, Socket socket)
  {
    // Ens subscrivim als canvis
    conversacio.addListDataListener(this);
  }

  // Tractem el canvi de qualsevol cosa afegida al DefaultListModel.
  public void intervalAdded(ListDataEvent e)
  {
    // S'obté el text afegit
    String text = (String) conversacio.getElementAt(e.getIndex0());
    // S'envia pel socket
    output.writeUTF(text);
  }
}
```

- Usar el patró observador com hem aprés en la primera pràctica, és una forma elegant orientada a objectes que permet evitar que hi hagi algú que sàpiga tots els fils que hi ha en marxa i haver-los d'avisar un a un cada vegada (és una diferència amb la nostra implementació de la pràctica 2!! On teníem un bucle i passàvem el text a cada fil 1 per 1!) i així evitar tenir una llista amb tots els fils i actualitzant-los un per un.

Client

Pel que fa el client, està estructurat amb la conversació, que és la nostra base de dades i que els clients no s'encarreguen de modificar (podríem suposar que es el model del patró MVC). Després tenim una vista, que presenta tot, i s'assembla força en estructura a la vista de qualsevol MVC, ja que aquesta vista no manté cap gestió del exterior ni en sap res, no necessita cap **socket**. Aquest admet nous clients a la imatge (observar) i així s'assabenta d'events com el botó enviar en el camp de text i al mateix temps pot aconseguir texts d'usuaris i afegir-los en el textArea.

Un altre aspecte que segueix les pautes del patró MVC és el nostre ClientControler, que òbviament com diu el nom, farà de controlador, és a dir, s'encarregarà de gràcies al socket associat, subscriure's al event **enviar**, afegint el text escrit al socket i al mateix temps tot el que arriba pel socket, l'afegeix a la vista del MVC.

Pel que fa el codi, he preferit que estigui molt comentat enlloc d'escriure pautes per aquí perquè és difícil generalitzar sobre aquest.

Aquí teniu unes quantes imatges del resultat final:

Patró Reactor

La manera més intuïtiva d'implementar un servidor multi-thread és seguir l'enfocament de thread per connexió. És adequat per a llocs que necessiten evitar la connexió per a compatibilitat amb biblioteques que no siguin segures.

També utilitza els millors mòduls de processament múltiple per aïllar cada sol·licitud, de manera que un problema amb una sola sol·licitud no afecti a cap altre.

Els processos són massa pesats, amb un canvi de context més lent i un major consum de memòria. Per tant, s'utilitza l'enfocament per connexió per millorar l'escalabilitat, tot i que la programació amb fils és susceptible d'error i és difícil de depurar.

Per tal d'acordar el nombre de fils per obtenir el millor rendiment general i evitar la sobrecàrrega de creació / destrucció de fils, és una pràctica habitual posar un sol subministrador d'enviament davant d'una cua de bloqueig delimitada i a un conjunt de fils. El distribuïdor bloqueja en el sòcol les noves connexions i les ofereix a la cua de bloqueig delimitada. Les connexions que superin la limitació de la cua es reduiran, però les latències per a les connexions acceptades es tornen previsibles. Un conjunt de subhastes fa una enquesta a la cua de les sol·licituds entrants, que es processaran i respondran.

Desafortunadament, sempre hi ha una relació individualitzada entre connexions i fils. Les connexions de llarga durada com les connexions Keep-Alive donen lloc a un gran nombre de fils de treballadors que esperen en un estat d'espera, p. Ex. accés al sistema de fitxers, xarxa, etc. A més, centenars o fins i tot milers de fils concurrents poden perdre una gran quantitat d'espai de pila a la memòria.

Arquitectura dirigida per esdeveniments
L'enfocament orientat a esdeveniments pot separar els fils de les connexions, que només utilitzen fils per a esdeveniments en devolucions o gestors específics.

Una arquitectura orientada a esdeveniments consisteix en creadors d'esdeveniments i consumidors d'esdeveniments. El creador, que és la font de l'esdeveniment, només sap que l'esdeveniment s'ha produït. Els consumidors són entitats que necessiten conèixer l'esdeveniment. Poden estar implicats en processar l'esdeveniment o simplement es veuran afectats per l'esdeveniment.

Què és?

El patró del reactor és una tècnica d'implementació de l'arquitectura orientada a esdeveniments. En termes simples, utilitza un bucle d'events d'un sol threaded en esdeveniments que emeten recursos i els envia als corresponents gestors i devolucions de trucades.

No hi ha necessitat de bloquejar els E / S, sempre que els gestors i devolucions de trucades dels esdeveniments estiguin registrats per tenir-ne cura. Els esdeveniments es refereixen a instàncies com una nova connexió entrant, preparades per a la lectura, llestes per escriure, etc. Aquests controladors / callbacks poden utilitzar un pool de subprocesos en entorns multi-nucli.

Aquest patró dissocia el codi de nivell d'aplicació modular de la implementació del reactor reutilitzable.

Parts

Hi ha dos participants importants en l'arquitectura del patró del reactor:

1. Reactor

Un reactor funciona en un fil separat, i el seu treball és reaccionar als esdeveniments IO enviant el treball al gestor adequat. És com un operador de telefonia en una empresa que respon les trucades dels clients i transfereix la línia al contacte adequat.

2. Handlers

Un handler realitza el treball real que es farà amb un esdeveniment d'E / S, similar a l'oficial actual de la companyia amb què el client vol parlar.

Un reactor respon als esdeveniments d'E / S enviant el gestor adequat. Els handlers realitzen accions que no bloquegen.

El patró arquitectònic de Reactor permet aplicacions orientades a esdeveniments per desmultiplexar i enviar sol·licituds de servei que es lliuren a una aplicació d'un o més clients.

Un reactor seguirà buscant esdeveniments i informarà el controlador d'esdeveniments corresponent per gestionar-lo quan es produeixi un esdeveniment.

Comentaris

El patró del reactor és un patró de disseny per al desmultiplexació síncrona i l'ordre dels esdeveniments a mesura que arriben.

Rep missatges, sol·licituds i connexions procedents de diversos clients concurrents i processen aquests missatges de manera seqüencial mitjançant gestors d'esdeveniments. El propòsit del patró de disseny del reactor és evitar el problema comú de crear un fil per a cada missatge, sol·licitud i connexió. A continuació, rep els esdeveniments d'un conjunt de controladors i els distribueix de manera seqüencial als gestors d'esdeveniments corresponents.

Aquesta descripció sona similar a la descripció donada a un descodificador, però s'utilitza per seleccionar entre molts dispositius, mentre que un desmultiplexor s'utilitza per enviar un senyal entre molts dispositius.

Per què importa?

Els servidors han de gestionar més de 10.000 clients concurrents i els fils no poden escalar les connexions mitjançant Tomcat, Glassfish, JBoss o HttpClient.

Per tant, l'aplicació que utilitza el reactor només ha d'utilitzar un fil per gestionar esdeveniments simultanis. Bàsicament, el reactor estàndard permet una aplicació de plom amb esdeveniments simultanis, mantenint la senzillesa de la rosca única.

Un desmultiplexor és un circuit que té una entrada i més d'una sortida. És un circuit que s'utilitza quan es vol enviar un senyal a un dels diversos dispositius.

Com que el patró del reactor és usat per Node.js, Vert.x, Reactive Extensions, Jetty, Ngnix i altres. Així, si es vol saber com funcionen les coses darrere de l'escena, és important parar atenció a aquest patró.

Bibliografia:

http://www.chuidiang.org/java/sockets/hilos/socket_hilos.php

https://gl-epn-programacion-ii.blogspot.com/2010/04/que-es-swing.html

http://www3.uji.es/~belfern/Docencia/Presentaciones/ProgramacionAvanzada/Tema3/swing2.html#32