Sprawozdanie nr 7

Data ćwiczeń: 14.05.2015

Data oddania sprawozdania: 12.06.2015

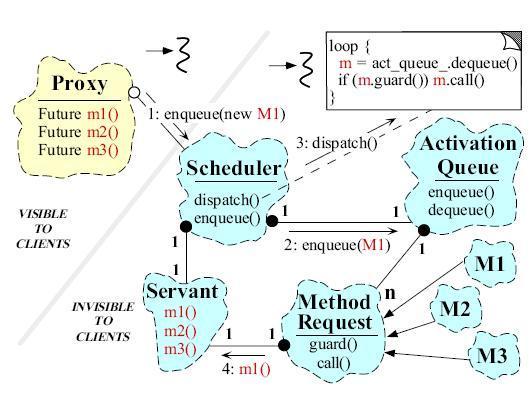
* Tematyka

Na siódmych zajęciach laboratoryjnych podejmowaliśmy wyzwanie zaimplementowanie rozwiązania problemu producentów i konsumentów z wykorzystaniem wzorca projektowego Active Object, specjalnie przeznaczonego do problemów z synchronizacją wątków i blokowaniem dostępu do bufora.

* Zadanie

Na zajęcia należało zaimplementować problem producentów i konsumentów z wykorzystaniem wzorca Active Object. W treści zadania zdefiniowane było, że to bufor w tym zadaniu powinien pełnić rolę Active Object. Było to jedno z trudniejszych zadań, z którym musieliśmy poradzić sobie na zajęciach z teorii współbieżności, aczkolwiek mieliśmy na nich okazję poznać najpopularniejszy wzorzec związany z tą tematyką.

W skrócie wzorzec ten wymusza na programiście takie zaprojektowanie systemu, aby oddzielone zostało wykonanie metody od jej wywołania. Dzięki zewnętrznemu, zaimplementowanemu przez nas *Schedulerowi*, poprawia się współbieżność programu i upraszcza synchronizowany dostęp do obiektów, które są umieszczone w swoich własnych wątkach.



* Kod zadania

*/\*\*  
 \* Created by Patryk on 2015-05-14.  
 \*/***public class** Main {  
 **private final static int *PROD\_COUNT*** = 3;  
 **private final static int *CONS\_COUNT*** = 5;  
 **private final static int *ACTIVE\_OBJECTS*** = 10;  
  
 **public static void** main(String[] argv) **throws** InterruptedException {  
 ActiveObject activeObject = **new** ActiveObject(***ACTIVE\_OBJECTS***);  
 Proxy proxy = activeObject.getProxy();  
 Producer[] producers = **new** Producer[***PROD\_COUNT***];  
 Consumer[] consumers = **new** Consumer[***CONS\_COUNT***];  
  
 **for** (**int** i = 0; i < ***PROD\_COUNT***; i++) {  
 producers[i] = **new** Producer(i, proxy);  
 }  
  
 **for** (**int** i = 0; i < ***CONS\_COUNT***; i++) {  
 consumers[i] = **new** Consumer(i, proxy);  
 }  
  
 **for** (**int** i = 0; i < ***CONS\_COUNT***; i++) {  
 consumers[i].start();  
 }  
  
 **for** (**int** i = 0; i < ***PROD\_COUNT***; i++) {  
 producers[i].start();  
 }  
  
 **for** (**int** i = 0; i < ***PROD\_COUNT***; i++) {  
 producers[i].join();  
 }  
  
 **for** (**int** i = 0; i < ***CONS\_COUNT***; i++) {  
 consumers[i].join();  
 }  
 }  
}

**public class** ActiveObject {  
 **private** Buffer **buffer**;  
 **private** Scheduler **scheduler**;  
 **private** Proxy **proxy**;  
  
 **public** ActiveObject(**int** queueSize){  
 **super**();  
 **buffer** = **new** Buffer(queueSize);  
 **scheduler** = **new** Scheduler();  
 **proxy** = **new** Servant(**buffer**, **scheduler**);  
 **scheduler**.start();  
 }  
  
 **public** Proxy getProxy(){  
 **return this**.**proxy**;  
 }  
}

**public class** AddRequest **implements** IMethodRequest {  
 **private** Buffer **buffer**;  
 **private** Object **object**;  
  
 **public** AddRequest(Buffer buffer, Object object){  
 **this**.**buffer** = buffer;  
 **this**.**object** = object;  
 }  
  
 @Override  
 **public void** call() {  
 **buffer**.add(**object**);  
 }  
  
 @Override  
 **public boolean** guard() {  
 **return** !**buffer**.isFull();  
 }  
}

**public class** Buffer {  
 **private int bufSize**;  
 **private** Queue<Object> **buffer**;  
 **public** Buffer(**int** bufSize){  
 **this**.**bufSize** = bufSize;  
 **this**.**buffer** = **new** LinkedList<Object>();  
 }  
 **public void** add(Object object) {  
 **if**(!**this**.isFull()){  
 **this**.**buffer**.add(object);  
 }  
 }  
 **public** Object remove() {  
 **if**(**this**.isEmpty()){  
 **return null**;  
 }  
 **else**{  
 **return buffer**.remove();  
 }  
 }  
 **public boolean** isFull() {  
 **return buffer**.size() == **bufSize**;  
 }  
 **public boolean** isEmpty() {  
 **return buffer**.isEmpty();  
 }  
}

**public class** CustomFuture {  
 **private** Object **object**;  
 **public void** setObject(Object object){  
 **this**.**object** = object;  
 }  
 **public** Object getObject(){  
 **return object**;  
 }  
 **public boolean** isReady(){  
 **return object** != **null**;  
 }  
}

**public interface** IMethodRequest {  
 **boolean** guard();  
 **void** call();  
}

**public interface** Proxy {  
 **void** add(Object object);  
 CustomFuture remove();  
}

**public class** RemoveRequest **implements** IMethodRequest {  
 **private** Buffer **buffer**;  
 **private** CustomFuture **customFuture**;  
  
 **public** RemoveRequest(Buffer buffer, CustomFuture customFuture){  
 **this**.**buffer** = buffer;  
 **this**.**customFuture** = customFuture;  
 }  
  
 @Override  
 **public void** call() {  
 **customFuture**.setObject(**buffer**.remove());  
 }  
  
 @Override  
 **public boolean** guard() {  
 **return** !**buffer**.isEmpty();  
 }  
}

**public class** Scheduler **extends** Thread {  
 **private** Queue<IMethodRequest> **activationQueue**;  
  
 **public** Scheduler() {  
 **activationQueue** = **new** ConcurrentLinkedQueue<IMethodRequest>();  
 }  
  
 **public void** enqueue(IMethodRequest request) {  
 **activationQueue**.add(request);  
 }  
  
 **public void** run() {  
 **while** (**true**) {  
 IMethodRequest imr = **activationQueue**.poll();  
 **if** (imr != **null**) {  
 **if** (imr.guard()) {  
 imr.call();  
 } **else** {  
 **activationQueue**.add(imr);  
 }  
 }  
 }  
 }  
}

**public class** Servant **implements** Proxy {  
 Buffer **buffer**;  
 Scheduler **scheduler**;  
  
 **public** Servant(Buffer buffer, Scheduler scheduler){  
 **this**.**buffer** = buffer;  
 **this**.**scheduler** = scheduler;  
 }  
  
 @Override  
 **public void** add(Object object){  
 **scheduler**.enqueue(**new** AddRequest(**buffer**, object));  
 }  
  
 @Override  
 **public** CustomFuture remove(){  
 CustomFuture customFuture = **new** CustomFuture();  
 **scheduler**.enqueue(**new** RemoveRequest(**buffer**, customFuture));  
 **return** customFuture;  
 }  
}

**public class** Consumer **extends** Thread{  
 **private int id**;  
 **private** Proxy **proxy**;  
  
 **public** Consumer(**int** id, Proxy proxy){  
 **this**.**id** = id;  
 **this**.**proxy** = proxy;  
 }  
  
 **public void** run(){  
 **while**(**true**){  
 CustomFuture consumed = **proxy**.remove();  
 **while**(!consumed.isReady()){  
 System.***out***.println(**"Konsument "** + **id** + **" czeka."**);  
 **try** {  
 Thread.*sleep*(500);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.***out***.println(**"Konsument "** + **id** + **" zjadl: "** + consumed.getObject());  
 **try** {  
 Thread.*sleep*(500);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

**public class** Producer **extends** Thread{  
 **private int id**;  
 **private** Proxy **proxy**;  
 **private** Random **rand**;  
  
 **public** Producer(**int** id, Proxy proxy){  
 **this**.**id** = id;  
 **this**.**proxy** = proxy;  
 **rand** = **new** Random();  
 }  
  
 **public void** run(){  
 **while**(**true**){  
 **int** tmp = **rand**.nextInt(100);  
 **proxy**.add(tmp);  
 System.***out***.println(**"Pracownik "** + **id** + **" dodal: "** + tmp);  
 **try** {  
 Thread.*sleep*(200);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}