**Observer Tasarım Kalıbı**

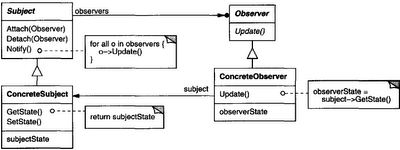
**Tür:** *Davranışsal Şablon*

**Diğer Adı:***Yayınla-Abone Ol (Publish-Subscribe)*

**İzleyici Şablonu (*Observer Pattern*)** öyle bir birden çoğa (*one-to-many*) ilişki tanımlar ki; bir nesne (konu [*subject*] yada izlenilen [*observable*]) değiştiği zaman ona kayılı olan (*registered*) tüm nesneler (izleyiciler [*observers*]) haberdar edilir (*notified*).

İzleyici şablonun amacı, belirli bir konuda yayın yapan kaynağın, bu bilgiye ihtiyaç duyan nesneler tarafından takip edilmesini sağlayacak bir arayüz sağlamaktır. Bu yüzden Yayınla-Abone Ol Şablonu da denir.





Yukarıda bu şablonun sınıf şemasını görüyoruz. Subject izlenilen, Observer ise izleyen nesneler için arayüz (*interface*) sınıflarıdır. Bu sınıflar standard Java kütüphanesinde de [*Observable*](http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Observable.html)ve [*Observer*](http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Observer.html) olarak yer alırlar.

Bir sınıfı yayıncı yada izleyici yapmak için ilk önce gerekli arayüzlere uyması (*implement*) gerekir. Ardından gereklli metotlar yeniden yazılarak (*override*) bu çerçeveye (*framework, paradigm*) uyulmuş olunur.

İzleyici Şablonunun kullanım alanları geniştir. Görsel uygulamalarda eğer görüntü (*view*) ve iş mantığı (*business logic*, burada *model*) ayrıysa (yani *MVC*Şablonuna uyuluyorsa) görüntü modelin değiştiğini ona abone olarak takip edebilir. Örnek olarak, ekranda bir düğmeye basıldığında modele bir sinyal gönderilip, bu olay (*event*) işlenebilir yada modelde bir katarın (*string*) değişmesiyle görüntüde anında bu katar ile bağlanmış görsel yinelenebilir. Birden fazla görüntü tek bir haber kaynağına da üye olabilir. Bir başka uygulama ise haber takibi yada RSS gibi takip sistemleri olabilir.

Bilindiği gibi *Twitter*'da kullanıcılar (*contact*) bir birlerini takip ederler (*follow*) yada takip etmeyi bırakabilirler (*unfollow*). Birisi bir mesaj yazdığında (*tweet*) onu takip eden tüm kullanıcılar bu mesajı alacaklardır. Dolayısıyla burada konu da takipçi de kullanıcı sınıfı olacaktır, fakat takip eden ve edilen sınıf nesneleri (veya örnekleri [*instance*]) farklı olacaktır. Bu durum bu örnek için bu şekilde denk gelmiştir. Genellikle takip eden ve edilen sınıflar farklı olurlar.   
  
İlk olarak, *Observable*arayüzünün kaynak kodlarını inceleyelim:

|  |
| --- |
| public interface Observable {    void registerObserver(Observer observer);  void removeObserver(Observer observer);  void notifyObservers(Object data);  } |

Tahmin edileceği gibi, registerObserver Observer arayüzüne uyan sınıf türünden nesneleri izlenilen nesneye kaydeder. removeObserver metodu ise kaydolan izeleyici nesnesini izleyiciler listesinden siler. notifyObservers Java standard kütüphanesinden farklı olarak, Observable türünden bir operandı  mevcut değildir çünkü burada Object sınıfı yeterince genel amaçlı olduğu için, izleyici sınıfların izledikleri sınıfın içeriğini bilmesi gerekmez.

|  |
| --- |
| public interface Observer {        public void update(Object data);  } |

**update**metodu *Observable*arayüzünün **notifyObservers**metodu tarafından çağrılır.

|  |
| --- |
| public class Message {        // ctors        private String getDateText() {          // ...      }      @Override      public String toString() {          return textBody + " by " + sender + " @ " + getDateText();      }      // getters and setters        private String textBody;      private String sender;      private Timestamp dateSent;  } |

*Message*sınıfı iki kullancı arasındaki mesajlaşma verisini temsil eder. Mesajın yazısı **textBody**, gönderenin ismi **sender**(burada gönderen isminin tekil [unique] olduğu kabul edilmiştir) ve gönderildiği tarih **dateSent**, değişkenleri ile temsil edilmiştir. **getDateText**metodu o andan mesajın gönderildiği zaman arasındaki farkı özel bir biçimde yazı olarak döner. **getDateText**metodunun ayrıntılarına kaynak koddan bakabilirsiniz.

|  |
| --- |
| public class Contact implements Observable, Observer, Iterable<Message> {      // ctors          @Override      public void update(Object data) {          addMessage((Message) data);      }      @Override      public void registerObserver(Observer observer) {          observingContacts.add((Contact) observer);      }      @Override      public void removeObserver(Observer observer) {          observingContacts.remove(observer);      }      @Override      public void notifyObservers(Object data) {          for (Contact notifiedContact : observingContacts) {              notifiedContact.update(data);          }      }      public void tweet(String textBody) {          Message message = new Message(textBody, name);          addMessage(message);          notifyObservers(message);      }      public void addMessage(Message message) {          messages.add(message);      }      // getters and setters        @Override      public Iterator<Message> iterator() {          return messages.iterator();      }      private String name;      private ArrayList<Message> messages;      private ArrayList<Contact> observingContacts;  } |

*Contact*sınıfı ise en temel model sınıfıdır. Sistemdeki kullancıları temsil eder. Kullanıcının ismi, duvarındaki mesajlar (**messages**) ve o kullanıcıyı takip eden diğer kullanıcı nesneleri (**observingContacts**) bu sınıfın alanlarıdır (*fields*). Bir kullanıcı güncellendiğinde ona mesaj gelmiş demektir. Bu anlaşma daha önceden somut (*concrete*) sınıflar arasında yapılmalıdır (burada iki sınıf da *Contact*olduğu için tüm metodlar bu sınıfta yer almıştır). **registerObserver**metodu **observingContacts**listesine izleyicileri ekler, **removeObserver**metodu ise bu listeden siler.**notifyObservers**metodu da *foreach*döngüsü içinde tüm izleyen nesnelerin **update**metodlarını çağırmak suretiyle onları haberdar eder.**tweet**metodu mesajı yazan kullanıcının adı ve yazısıyla yeni bir mesaj nesnesi oluşturup mesaj listesine ekler ve işte tam da burada, anında, abonelere bu mesaj bildirimi gönderilir.  
  
*Contact*sınıfındaki **iterator**metodu *Iterator*(Yineleme) Şablonu ile ilgilidir. Ayrıntısı ayrıca bir yayın olarak bahsedilecektir, fakat kısaca; bu metot, *foreach*döngüsü ile kişi nesnesinin üzerinden mesajlarını teker teker alınmasını sağlar. Bu kullanıcı (*client*) kodda büyük kolaylığa yol açacaktır.  
  
Son olarak aşağıda kullanıcı program parçasını görebilirsiniz.

|  |
| --- |
| public class TwitterClient {      public static void main(String[] args) {            // Creating contacts          Contact harold = new Contact("Harold");          //...            // Adding contacts to current session          session.add(harold);          //...          session.add(yigit);          // Registering Harold, Jennifer, Hussein and Aliyyah as 'Observers' of          // Yigit, e.g. now they follow Yigit          yigit.registerObserver(harold);          //...          // Some random tweets          try {              jennifer.tweet("I am bored today");              Thread.sleep(1000);              hussein.tweet("I'll be there in a minute...");              Thread.sleep(1000);              yigit.tweet("Hi everyone! It's a nice shiny day out there!");              Thread.sleep(1000);              jennifer.tweet("It is ok!");              Thread.sleep(3000);          } catch (InterruptedException e) {              e.printStackTrace();          }          // Print walls          System.out.println("Everyone follows Yigit...");          System.out.println();          printWalls(session);          // Remove Harold and Aliyyah from Yigit's observers list, e.g. now          // Harold and Aliyyah unfollowed Yigit          yigit.removeObserver(harold);          yigit.removeObserver(aliyyah);          try {              yigit.tweet("Harold and Aliyyah cannot receive what I write, so I'm cool");              Thread.sleep(1500);          } catch (InterruptedException e) {              // TODO Auto-generated catch block              e.printStackTrace();          }          // Print walls again          System.out.println("Harold and Aliyyah unfollowed Yigit. How mean of them!");          System.out.println();          printWalls(session);      }      public static void printWalls(ArrayList<Contact> session) {          for (Contact contact : session) {              System.out.println(contact.getName() + "'s wall:");              for (Message message : contact) {                  System.out.println(message);              }              System.out.println();          }      }  } |

Yukarıdaki örnekte hayali bir oturumda kullanıcılar eklenir. Ardından, herkes Yiğit adlı kullanıcıya abone olurlar. Herkes kendi attığı mesajı duvarında görür. Ayrıca Yiğit mesaj attığında Yiğit'in de mesajını göreceklerdir. Ardından Harold ve Aliyyah Yiğit'i takip etmeyi bırakırlar. Bundan sonra artık Yiğit mesaj yazdığında onlara bu mesaj gitmez. Oturumdaki tüm kişilerin mesajlarını taramak ve yazdırmak için **printWalls**metodu yazıldı. Bu metotta her kullancı oturumdan elde edilir ve daha önce bahsettiğim Iterator yapısı ile de bu kişinin sahip olduğu mesajlar okunur ve konsola yazılır.  
  
**NOT:** Bu kod parçasında zaman farkından kaynaklanan süreleri simule etmek için araya Thread sınıfının sleep metodu ile gecikmeler konuldu.