Design Patterns: Observer

Sinan GÖKÇE

Observer Pattern – Nedir?

Observer pattern en popüler design patternlerden birisidir. «Notification» ve «Achivement» sistemleri için kullanılanır.

Observer pattern, bir sistem içerisindeki farklı bölümlerin birbirine engel olmadan çalışmasını sağlar.

Observer Pattern – Nedir?

Örneğin; Oyunda karakterin köprüden atlaması bir başarı ise oyunun fizik motorunda hiçbir kod değişikliğine gerek duyulmadan sadece observer listesindeki gözlemcileri kontrol ederek fizik motoru ile haberleşmesini ve başarının gerçekleşip gerçekleşmediğini belirleyerek programcıya bunu istediği gibi kullanma seçeneği verir.

```
void Physics::updateEntity(Entity& entity)
{
  bool wasOnSurface = entity.isOnSurface();
  entity.accelerate(GRAVITY);
  entity.update();
  if (wasOnSurface && !entity.isOnSurface())
  {
    notify(entity, EVENT_START_FALL);
  }
}
```

```
class Observer
{
public:
  virtual ~Observer() {}
  virtual void onNotify(const Entity& entity, Event event) = 0;
};
```

Observer Pattern – Nedir?

```
class Achievements : public Observer
{
public:
  virtual void onNotify(const Entity& entity, Event event)
  {
    switch (event)
```

```
    {
    case EVENT_ENTITY_FELL:
      if (entity.isHero() && heroIsOnBridge_)
      {
        unlock(ACHIEVEMENT_FELL_OFF_BRIDGE);
      }
      break;

      // Handle other events, and update heroIsOnBridge_...
    }
  }

private:
  void unlock(Achievement achievement)
  {
    // Unlock if not already unlocked...
  }

  bool heroIsOnBridge_;
};
```

//Yandaki kod örneğinde achivement sistemini görüyoruz. Kahramanımızın köprüden düşüp düşmediği kontrol edilerek, eğer düştüyse unlock fonksiyonu çalıştırılarak bu başarımın açılması(eğer daha önce açılmadıysa) sağlanmıştır.

Observer Pattern – Nedir?

Bu kalıp one-to-many olayını destekleyen tasarım desenidir. Bir nesnenin değişikliğinden farklı nesneler etkilenecek ise bu kalıp tavsiye edilir.

Örneğin; bir alışveriş sitesinde bir ürüne indirim yapıldığında kullanıcılarınıza e-mail ile haber verilir iken bu kalıp kullanılabilir. Ya da en basitinden facebook da bir gruba üyesiniz grupta bildirimleri açtığınızda size (ve daha birçok kişiye) gelecek olan bildirim bu yapı ile olabilir.

Observer Pattern – Nedir?

Observer listesini istediği gibi değiştirebilmek ve yeni başarılar ekleyip çıkarmak, yazılan kod için esneklik(flexibility) sağlar. Bu da programlamada temel ilkelerden biridir.

Modularity..

Subject – Nedir?

Subject(özne), gözlemlenecek durumu bizzat yapan(fail)dır.

Observer listesini tutmak ve bildirim göndermek üzere iki görevi bulunur.

Subject observer listesi tutar fakat oraya bağlı değildir. Sadece achivement sistemi ile haberleşir.

Subject – Nedir?

- Observer'ın liste şeklinde olması önemlidir. Çünkü aynı olayı gözlemleyen birden fazla observer olabilir.

- Örneğin; kahramanımız köprüden atladığında bu event için bir rozet verilirken, suya düştüğünde ses motoru da ona uygun bir ses üretir. Eğer liste değil de bir tane olsaydı, bu iki durumu aynı anda gerçekleştirmezdik.

```
class Subject
{
public:
  void addObserver(Observer* observer)
  {
    // Add to array...
  }

  void removeObserver(Observer* observer)
  {
    // Remove from array...
  }

  // Other stuff...
};
```

Observable Physics

- Physics sınıfına subject sınıfı inherit edilerek çalışır. İnheritance ile Subject sınıfına ait metodlar «protected» olarak kullanılabilir.

```
class Physics : public Subject
{
public:
  void updateEntity(Entity& entity);
};
```

- Bununla birlikte fizik motoru(physics engine) bildirmeye değer bir şey farkettiğinde «notify()» metodunu kullanarak bildirimde bulunabilir.

«Yavaş mı?»

- Observer pattern'ı 'eventler' ve 'mesajlar' gibi karakterleri barındırdığından dolayı kötü bir üne sahiptir.
- Bazı sistemlerin yavaş olması işlemlerin kuyruklama ve dinamik(değişken) hafıza ayırmasından dolayıdır.
- Ama yukarıda bu pattern'in nasıl uygulandığını gördük; notification sistemi basitçe bir listeyi dolaşmak ve bazı sanal metodları çağırmaktan ibarettir.
- Bazı metodların yavaş çalışması, kritik kodlardaki yüksek performansından dolayı göz ardı edilebilir.

«Çok mu hızlı?»

- Esasında observer sınıfı senkronize olduğu için çok dikkatli olunmalıdır; subject sınıfı, gözlemcilerin hepsini dolaştığı için hepsinin bildirim vermesini bekler. Yavaş çalışan bir gözlemci Subject'i tıkayabilir.
- Eğer bir duruma senkronize cevap verilirse, kullanıcı ara yüzünün tıkanmaması için, işleri hızlıca halledebilmek gerekir. Eğer işler yavaşsa başka bir «thread» kullanılması gerekir.

«Çok Fazla Dinamik Hafıza Kullanması»

- Dinamik hafıza, üzerinde ekleme-çıkarma işlemlerini yapabilmek için makine tarafından hafızada yer ayrılmasıdır.
- Yukarıdaki örnekte observer listesini array olarak tanımladık(örneği kolaylaştırmak için), gerçek kodda bu dinamik hafıza şeklinde kullanılmalıdır(observer ekleme-çıkarma işlemini kolay yapabilmek için).

«Çok Fazla Dinamik Hafıza Kullanması»

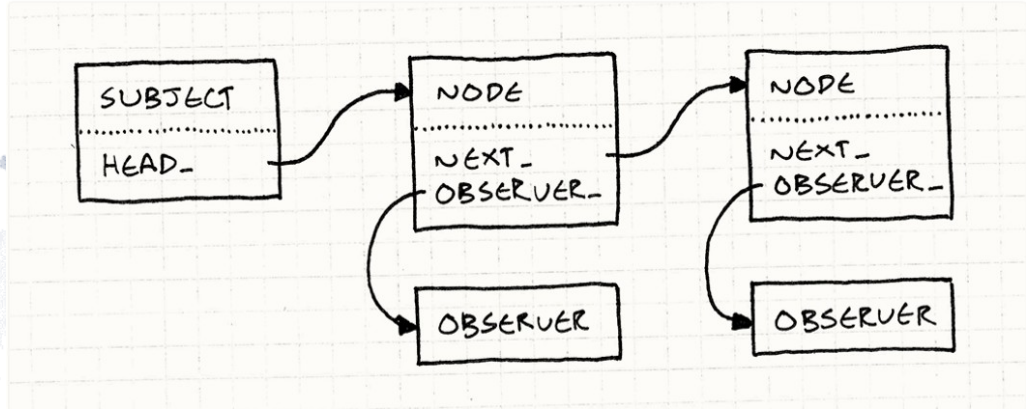
- Dinamik hafızadan korkulmasının sebebi; hafızada ayrılan bölüm sabit olmadığı için bir hata sonucu hafızanın gereksiz yere doldurulması(Nasa'nın ay görevi sırasında yaşadığı bilgisayar hatası: https://www.youtube.com/watch?v=z4cn93H6sM0) ve sistemin çökmesi ihtimalidir.
- Fakat bu durum, eskisi kadar korkulan bir durum değildir. Çöp toplayıcı(Garbage Collector) hafızadaki ölü durumda olan yani kullanılmayan objeleri ve değişkenleri hafızadan otomatik siler.

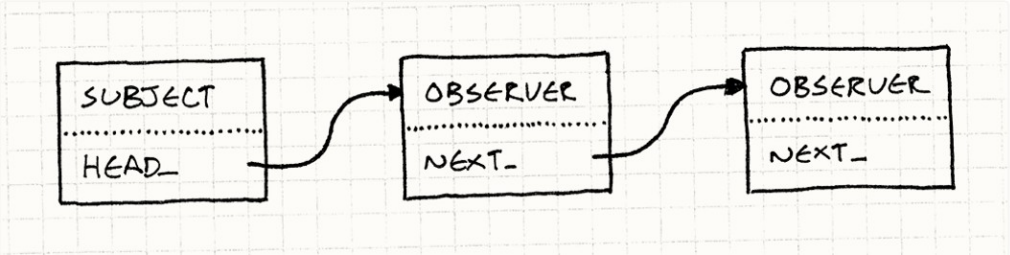
«Çok Fazla Dinamik Hafıza Kullanması»

- Dinamik hafıza önceki verilen kod örneklerinde sadece observer listesi için kullanıldı. Bildirim gönderme hafıza ayırmaya ihtiyaç duymaz, sadece metod çağırma işidir.
- Dinamik hafıza ayırmaya oyunlar gibi kritik projelerde çokça ihtiyaç duyulabiliyor.

«Çok Fazla Dinamik Hafıza Kullanması»

- Dinamik hafıza konusunu yine de dert edinen ve kullanmaktan çekinen programcılar varsa, observerları linked list şeklinde, tekli veya ikili olmak üzere kullanabilir.





«What’s going on? 🏝»

- Şu ana kadar konuştuğumuz problemler observer pattern’ın bilinçli olarak yapmak istediği bir şeyden doğuyor: Yapılması gereken, kaynağı itibari ile birbirinden farklı kod parçalarını birbirinden ayırmak ve aralarında sadece iletişim sağlamaktır. Örneğin; fizik motoru ve notification sistemi (modülerlik)

Observer Metodunun Bugünü/Yarını

- Observer pattern’ı class ağırlıklı bir pattern’dır. Bugünlerde fonsiyonel programlamanın çok popüler olması, sadece bir bildirim almak için koca bir ara yüz ‘implement’ etmeyi pek estetik bulmaz.
- Bunun yerine fonksiyon tabanlı bir bildirim sistemi daha çok tercih edilir.

Observer Metodunun Bugünü/Yarını

- Event ve notification sistemi bugünlerde bir çok popüler dilde(C#, JavaScript vb) bazı protokol ve anahtar kelimelerle birlikte kullanımı kolaylaşmıştır.
- Örneğin; C#'da observer olarak kaydettiğiniz nesne «delegate» olarak geçer.
- JavaScript'te ise observer'lar «EventListener» protokolünü kullanan nesneler olabilirken, sadece metod ya da fonksiyon da olabilirler.

Observer Metodunun Bugünü/Yarını

- Observer tarzı sistemler uzun zamandır kullanılıyor olsa da henüz herkesin beğendiği, en efektif çözüm yöntemi bulunamamıştır.
- Bugünlerde en çok kullanılan sistemlerden birisi «data binding»tir. Fakat bu sistem çok yavaş ve karmaşık olduğu için oyun motorunun çekirdeğine uygulanamamaktadır.

Observer Metodunun Bugünü/Yarını

- Gelinen noktada eski ama iyi observer pattern'ı hala kullanılmayı bekliyor. Belki adında bugünkü gibi fantastik isimler('functional', 'reactive') olmasada…

İt's dead simple and it Works. (Robert Nystrom)

Dinlediğiniz için teşekkürler…