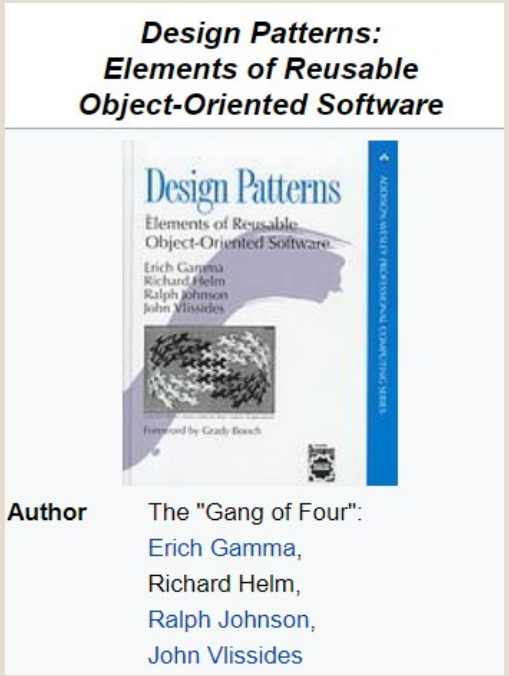
SİNGLETON

Diğer tasarım desenlerinden farklı olarak neler yapılmamanın anlatımı

Singleton

The Gang of Four tarafından tanımlanan Singleton paterni genellikle yarardan çok zarar verir. Desenin idareli olarak kullanılması gerektiğini vurgularlar, ancak bu mesaj oyun endüstrisine çeviride sıklıkla kayboldu.

Endüstrinin büyük bir kısmı C'den nesne yönelimli programlamaya (OOP) geçtiğinde karşılaştığı önemli sorunlardan biri "nasıl instance alabilirim?" idi. Aramak istedikleri bazı yöntemler vardı, ancak bu yöntemi elinde bulunduran nesnenin bir instance ı yoktu. Singletons (başka bir deyişle, küresel hale getirmek) kolay bir çıkış yoluydu.



Singleton

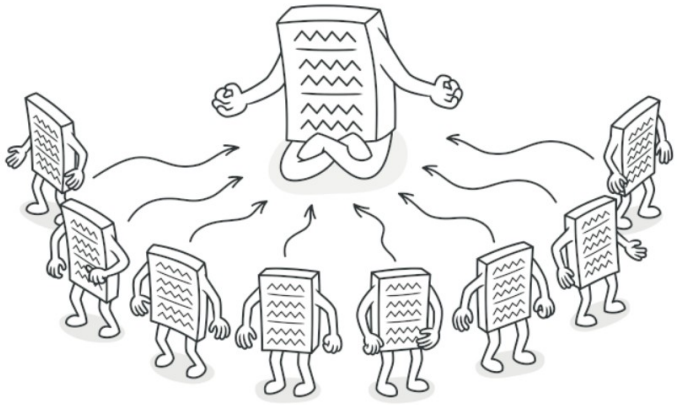
Tasarım Desenleri (Design Patterns) kitabında Singleton;

> *Ensure a class has one instance, and provide a global point of access to it.*

Yani,

- **bir sınıfın bir instance** ı olduğundan emin olun
- ona bir **erişim noktası** sağlayın

deniliyor.



SİNGLETON – BİR SINIFI BİR İNSTANCE İLE SINIRLANDIRMA

Singleton – bir sınıfı bir instance ile sınırlandırma

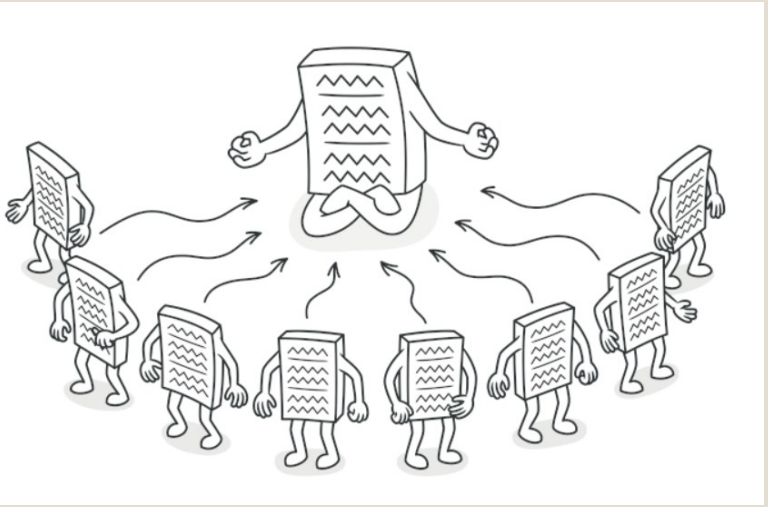
Örnek olarak bir API’de dosya işlemlerini paketleyen bir sınıf düşünebiliriz. Burada bir dosya oluşturma ve silme işlemlerini ele alabiliriz.

Bu 2 işlem asenkron olarak gerçekleşmek durumunda olacaktır. Çünkü, henüz yeni eklenmiş bir dosyayı kullanmadan silmek istemeyiz. O yüzden bu işlemler birbirleriyle koordineli çalışmak zorundalar.

Bunun sonucunda bir dosya oluşturma ve bir dosya silme çağrısı başlatılmalı ve paketleyicinin her iki çağrıyı da bilmesi gerekir.

Bunu yapmak için paketleyiciye yapılan bir çağrının önceki her işleme ait bilgiye sahip olması gerekir. Yazılımcı her ne kadar istediği kadar instance oluşturabilse de bu instance lar birbirlerinden bağımsızdır.

Bu yüzden singleton ile bir sınıfın sadece bir instance oluşturulması sağlanır.



SİNGLETON – KÜRESEL BİR ERİŞİM NOKTASI SAĞLAMAK

Singleton – küresel bir erişim noktası sağlamak

Örnek olarak oyunda günlük kayıt, içerik yükleme, oyun durumu kaydetme gibi sistemler, dosya sistemi paketleyicimizi kullanacak olsun. Bu sistemler eğer bizim paketleyicinin kendi instance larını oluşturamazsa başka nasıl elde edilirler?

Singleton ile tek bir instance oluşturmakla birlikte bunu **elde etmek için küresel olarak bir yöntem de sağlar**. Böylelikle oluşan instance herkes istediği yerden erişebilecek.

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    // Lazy initialize.
    if (instance_ == NULL) instance_ = new FileSystem();
    return *instance_;
  }

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

Singleton – küresel bir erişim noktası sağlamak

Yandaki örneğe göre,

static `instance_` üye, sınıfın bir instance ını tutar ve private yapı bunun tekil olmasını sağlar.

public `instance()` fonksiyon kod tarafında istenilen yerden bu `instance_` a erişimi sağlar.

Ayrıca `instance_` ilk kez çağırdığında singleton bunu otomatik olarak oluşturmakla görevlidir.

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    // Lazy initialize.
    if (instance_ == NULL) instance_ = new FileSystem();
    return *instance_;
  }

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

Singleton – küresel bir erişim noktası sağlamak

C ++ 11, yerel bir statik değişken (local static variable) için başlatıcısının senkron varlığında bile yalnızca bir kez çalıştırılmasını zorunlu kılar.

Modern bir C ++ derleyicisine sahip olduğunuzu varsayarsak, bu kod ilk instance olmadığı yerde iş parçacığı açısından güvenlidir.

Elbette, singleton sınıfın iş parçacığı güvenliği tamamen farklı bir sorundur! Bu sadece *başlatılmasını* sağlar .

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    static FileSystem *instance = new FileSystem();
    return *instance;
  }

private:
  FileSystem() {}
};
```

Singleton – Neden kullanıyoruz?

Dosya sistemi paketleyicimiz, ihtiyaç duyulan her yerde kullanılabilir durumdadır.

Sınıfın kendisi akıllıca birkaç instance ı somutlaştırarak bir şeyleri karıştırmamamızı sağlar.

Başka güzel özellikleri de var:

- ✓ **Hiç kimse kullanmazsa instance ı oluşturmaz.** Bellek ve CPU döngüleri kaydetmek her zaman iyidir. Singleton yalnızca ilk erişildiğinde başlatıldığından, oyun hiç istemezse veya sormazsa hiç instance oluşturulmayacaktır.

Singleton – Neden kullanıyoruz?

- ✓ **Çalışma zamanında başlatılır**. Singleton'a ortak bir alternative, statik üye değişkenleri olan bir sınıftır. Ancak statik üyelerin sahip olduğu bir sınırlama vardır: otomatik başlatma (automatic initialization). Derleyici statik main() çağrılmadan önce başlatılır.

Bu, yalnızca program çalışmaya başladıktan sonra bilinen bilgileri kullanamayacakları anlamına gelir (örneğin, bir dosyadan yüklenen konfigürasyon bilgileri). Ayrıca, birbirine güvenilir bir şekilde bağımlı olmayacakları anlamına gelir – derleyici, statiklerin birbirine göre başlatıldığı sırayı garanti etmez.

Tembel başlatma (Lazy initialization) her iki sorunu da çözer. Singleton mümkün olduğunca geç başlatılır ve o zamana kadar ihtiyaç duyulan her türlü bilginin mevcut olması sağlanır. Dairesel bağımlılıkları (circular dependency) olmadığı sürece, bir singleton kendisini başlatırken diğerine başvurabilir.

Singleton – Neden kullanıyoruz?

- ✓ **Singleton alt sınıflandırılabilir.** Bu güçlü ama çoğu zaman gözden kaçan bir özelliktir.

Örnek olarak dosya sistemimizin platformlar arası olması gerektiğini söyleyebiliriz. Bunu yapmak için, her platform için arabirimi uygulayan alt sınıflara sahip bir dosya sistemi için soyut bir ara birim olmasını istiyoruz. İşte temel sınıf

```
class FileSystem
{
public:
  virtual ~FileSystem() {}
  virtual char* readFile(char* path) = 0;
  virtual void  writeFile(char* path, char* contents) = 0;
};
```

Singleton – Neden kullanıyoruz?

Sonra birkaç platform için türetilmiş sınıfları tanımlarız:

```
class PS3FileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path)
  {
    // Use Sony file IO API...
  }

  virtual void writeFile(char* path, char* contents)
  {
    // Use sony file IO API...
  }
};

class WiiFileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path)
  {
    // Use Nintendo file IO API...
  }

  virtual void writeFile(char* path, char* contents)
  {
    // Use Nintendo file IO API...
  }
};
```

Sonra, FileSystem singleton a dönüşür:

```
class FileSystem
{
public:
  static FileSystem& instance();

  virtual ~FileSystem() {}
  virtual char* readFile(char* path) = 0;
  virtual void  writeFile(char* path, char* contents) = 0;

protected:
  FileSystem() {}
};
```

Singleton – Neden kullanıyoruz?

Önemli olan instance ın nasıl oluşturulduğudur:

```
FileSystem& FileSystem::instance()
{
  #if PLATFORM == PLAYSTATION3
    static FileSystem *instance = new PS3FileSystem();
  #elif PLATFORM == WII
    static FileSystem *instance = new WiiFileSystem();
  #endif

  return *instance;
}
```

Basit bir derleme anahtarı ile dosya sistemi paketleyicimize bağlanır. Yazılım tarafında `FileSystem::instance()` platforma özgü herhangi bir koda bağlanmadan dosya sistemine erişilir. Bu yapı FileSystem sınıfı içerisinde kapsüllenir.

Singleton – Neden kullandığımız için pişmanlık duyuyoruz?

Kısa vadede, Singleton modeli nispeten iyi huyludur.

Birçok tasarım seçeneği gibi, maliyeti uzun vadede ödüyoruz.

- **Küresel bir değişken (global state)**

Küresel değişkenler programın yapısını (structure), iş parçacıklarının (thread) durumunu ve yürütme akışını (execution) önemsemez

- **Kod hakkında akıl yürütmeyi zorlaştırırlar** (araya giren yeni bir iş ve hata bulma)
- **Bağlantıyı teşvik ederler**
- **Eşzamanlılık (senkron) dostu değiller**

Globallerin yarattığı sorunların bir listesi, bu desenin hiçbir problem çözmediği sonucuna vardırır. Çünkü singleton her ne kadar sınıflarda kapsüllenseler de hala global durumdalardır.

Singleton – Neden kullandığımız için pişmanlık duyuyoruz?

- **Sadece bir tane olsa bile iki problemi çözer**

İlk başta kulağa faydalı gibi görünse de eğer elimizde sadece bir problem varsa o zaman nasıl kullanacağız sorusu aklımıza takılmalı.

Tek instance sağlanması özelliği yararlı olabilecekken global erişimin sınırsız olarak yapılmasını istemeyebiliriz.

Benzer şekilde, global erişim de uygun olabilir ancak bu birden çok instance a izin veren bir sınıf için bile geçerli olan bir durum, değişim ihtiyacı duyulduğunda karmaşıklığa neden olabilecektir.

Beklenen çözüm olarak Log sınıfının tekil hale getirilebilir. Başlangıçta tek bir log dosyası kaydedilmek istendiğinde tek bir instance yeterli olacaktır. Ancak log a birden fazla dosyanın kaydetme işlemleri gerektiğinde bir sorunla karşılaşılacaktır. Birçok dosya yaratarak burayı bir çöplük haline getirilebilir. Yazılımcı, önem verilen singleton özelliğini bulmak için dosyalarda dolaşmak zorunda kalacak.

Singleton – Neden kullandığımız için pişmanlık duyuyoruz?

Bu durumu düzeltmek için log kaydetme işlemlerini gruplamamız gerekmektedir. Örnek olarak farklı oyun alanları için ayrı kaydediciler olacak; çevrimiçi, kullanıcı arayüzü, ses, oyun gibi. Ancak bunu bu şekilde yapamayız. Çünkü, log sınıfımız artık birden fazla instance oluşturmamıza izin vermekle kalmaz, aynı zamanda tasarım sınırlaması kullanan her bir arama kısmına da yerleşmiştir.

```
Log::instance().write("Some event.");
```

Log sınıfımızın birden çok anlık açmayı (başlangıçta yaptığı gibi) desteklemesi için hem sınıfın kendisini hem de ondan bahseden her kod satırını düzeltmemiz gerekir.Bizim için uygun olan erişim artık o kadar uygun değildir.

Singleton – Neden kullandığımız için pişmanlık duyuyoruz?

- **Tembel başlatma (lazy initialization), kontrolü bizden alır**

Sanal bellek ve yumuşak performans gereksinimleri masaüstü PC dünyasında, tembel başlatma akıllı bir hiledir.Oyunlar farklı bir türdedir.

Bir sistemin başlatılması zaman alabilir: bellek ayırma, kaynak yükleme, vb. Ses sistemini başlatma birkaç yüz milisaniye sürerse, bunun ne zaman olacağını kontrol etmeliyiz.Eğer bir sesin ilk çalınmasında kendini başlatmasına izin verirsek, bu oyunun aksiyon dolu bir bölümünün ortasında olabilir, bu da karelerin ve kekemeliğin düşmesine neden olabilir.

Aynı şekilde, oyunlarda genellikle parçalanmayı önlemek için bellek yığınının nasıl hazırlandığını yakından kontrol etmek gerekir. Eğer ses sistemimiz ilk devreye girdiğinde yığın ayırırsa, bu başlatmanın ne zaman gerçekleşeceğini bilmek isteriz, böylece o belleğin nerede yaşayacağını kontrol edebiliriz.

Singleton – Neden kullandığımız için pişmanlık duyuyoruz?

Bu iki sorun nedeniyle, görülen çoğu oyun tembel başlatma dayanmıyor. Bunun yerine, singleton desen böyle uygulanıyor :

```
class FileSystem
{
public:
  static FileSystem& instance() { return instance_; }

private:
  FileSystem() {}

  static FileSystem instance_;
};
```

Bu tembel başlatma sorununu çözer, ancak ham bir küresel değişken daha iyi yapmak birkaç singleton özellikleri atma pahasına. Statik bir örnekle, artık çok biçimliliği kullanamayız ve sınıf statik başlatma zamanında oluşturucu olmalıdır. Ne de instance gerekli olmadığında kullandığı bellek ücretsiz olabilir.

Bir singleton yaratmak yerine, burada gerçekten sahip olduğumuz şey basit bir statik sınıf. Bu mutlaka kötü bir şey de`Foo::bar()` `Foo::instance().bar()` yapısına göre daha basittir ve ayrıca statik bellekle gerçekten uğraştığınızı açıkça ortaya çıkarır.

SİNGLETON – YERİNE NE YAPABİLİRİZ?

Singleton – Yerine neler yapabiliriz?

→Bir sınıfın singleton özelliği ihtiyacı olup olmadığına karar verilir

```
class Bullet
{
public:
  int getX() const { return x_; }
  int getY() const { return y_; }

  void setX(int x) { x_ = x; }
  void setY(int y) { y_ = y; }

private:
  int x_, y_;
};

class BulletManager
{
public:
  Bullet* create(int x, int y)
  {
    Bullet* bullet = new Bullet();
    bullet->setX(x);
    bullet->setY(y);

    return bullet;
  }

  bool isOnScreen(Bullet& bullet)
  {
    return bullet.getX() >= 0 &&
           bullet.getX() < SCREEN_WIDTH &&
           bullet.getY() >= 0 &&
           bullet.getY() < SCREEN_HEIGHT;
  }

  void move(Bullet& bullet)
  {
    bullet.setX(bullet.getX() + 5);
  }
};
```

Kaç tane BulletManager instance ına ihtiyacımız olur?

0 !!

```
class Bullet
{
public:
  Bullet(int x, int y) : x_(x), y_(y) {}

  bool isOnScreen()
  {
    return x_ >= 0 && x_ < SCREEN_WIDTH &&
           y_ >= 0 && y_ < SCREEN_HEIGHT;
  }

  void move() { x_ += 5; }

private:
  int x_, y_;
};
```

Singleton – Yerine neler yapabiliriz?

→**Bir instance a rahat erişim sağlamak için**

Genel kural, işin yapılmasına rağmen değişkenlerin olabildiğince dar kapsamda tutulmasını istememizdir. Bir nesnenin kapsamı ne kadar küçük olursa, onunla çalışırken kafamızın içinde o kadar az yer tutmamız gerekir. Kod altyapısında bir nesneye erişmesinin diğer yollarını inceleyebiliriz :

- En basit ve genellikle en iyi çözüm, yalnızca ihtiyacınız olan işlevlere bir argüman olarak ihtiyacınız olan nesneyi oluşturmaktır.

- **Temel sınıftan alma** : Birçok oyun mimarisinin sığ ve geniş miras hiyerarşileri vardır ve sadece bir seviye derinliktedir.

Singleton – Yerine neler yapabiliriz?

Örneğin GameObject, oyundaki her düşman veya nesne için türetilmiş sınıfları olan bir temel sınıf olabilir. Bunun gibi mimarilerde, oyun kodunun büyük bir kısmı bu türetilmiş sınıflarda yaşayacaktır. Buradan tüm bu sınıfların zaten aynı şeye erişebileceği anlamı çıkarılıyor.

Burada, Log ,GameObject dışındaki hiçbir yapının nesneye erişimi olamayacağını gösteriyor. Ancak getLog(), türetilmiş her varlık için kullanılabilir.

```
class GameObject
{
protected:
  Log& getLog() { return log_; }

private:
  static Log& log_;
};

class Enemy : public GameObject
{
  void doSomething()
  {
    getLog().write("I can log!");
  }
};
```

Singleton – Yerine neler yapabiliriz?

- **Oluşturulmuş global nesneleri kullanma :** Kod tarafında çoğunlukla tüm oyun durumunu temsil eden tek bir nesne bulunur, örneğin tekil Game veya World gibi kullanılabilen birkaç nesne vardır. Global sınıfların sayısını azaltmak için varolan globaller kullanılmalıdır.

```
class Game
{
public:
  static Game& instance() { return instance_; }

  // Functions to set log_, et. al. ...

  Log&          getLog()         { return *log_; }
  FileSystem&   getFileSystem()  { return *fileSystem_; }
  AudioPlayer&  getAudioPlayer() { return *audioPlayer_; }

private:
  static Game instance_;

  Log         *log_;
  FileSystem  *fileSystem_;
  AudioPlayer *audioPlayer_;
};
```

Yandaki örnekte, Log, FileSystem ve AudioPlayer için ayrı ayrı singleton oluşturmak yerine tek bir global ile işimizi kolaylaştırırız.

Oluşturulan Game bir globaldir ve fonksiyonlar ile bunu diğer sistemler kullanabilir.

```
Game::instance().getAudioPlayer().play(VERY_LOUD_BANG);
```

Singleton – Yerine neler yapabiliriz?

Daha sonra mimari, birden çok Game instance ı (belki de akış veya test amacıyla), Log, FileSystem ve AudioPlayer'ı destekleyecek şekilde değiştirilirse, bu aradaki farkı bile anlayamayabilir. Bu durumdaki dezavantaj, daha fazla kodun Game global ine birleştirilmesiyle son bulur. Örnek olarak bir sınıfın sadece ses çalması gerekiyorsa, instance ses çalmaya ulaşmak için yine de global hakkında bilgi sahibi olması gerekecektir.

Bunu hibrit bir çözüm yoluyla ortadan kaldırabiliriz. Örneğin, varolan kodlarla Game üzerinden AudioPlayer a direk erişebiliyoruz. Varolmayanlar için bu yöntem çalışmamaktadır ama diğer yöntemler kullanılabilir.

- **Servis bulucudan (Service Locator) alma :** Şimdiye kadar, Game global sınıfının düzenli somut bir sınıf olduğunu varsaydık. Diğer bir seçenek olarak, tek varoluş nedeni nesnelere global erişim sağlamak olan bir sınıf tanımlamak olabilir. Bu ortak yapıya Servis Bulucu denir.

Singleton – Özetle;

- Singleton tasarım desenini kullanmak istiyorsak en az 2 kere düşünmeliyiz. Bu deseni kullanacaksak da, gerçekten ne kadar ihtiyacımızın var sorusunun cevabı önemli olmalıdır.

- Bu desenin, diğerlerine göre fazla veya gereksiz kullanımı bize faydadan çok zarar verir. Kısa vadeli olmasa da uzun vadede zararlı tarafı katlanarak büyüyebilir ve geri dönmek zorlaşır.

- Her oluşturulan instance ın bir global nesne veya değer olduğunu unutmayalım.

BCO-653 OYUN MİMARİSİ

IŞIK ESKİOĞLU

24.04.2020