KurumsalJava.com
Java Enterprise Architecture

Strategy - Strateji
Tasarım Şablonları Serisi

Özcan Acar
Bilgisayar Mühendisi
http://www.ozcanacar.com/

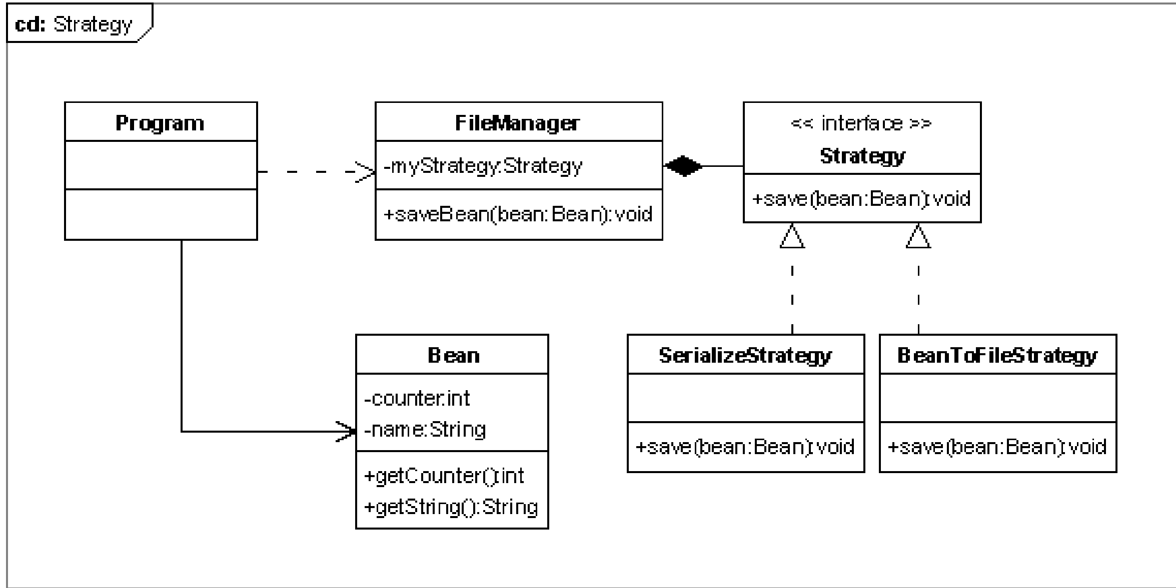
Bu Pdf dosya herhangi bir server üzerine kopyalanarak, indirime sunulabilir. Bunun için özel izin alınması gerekmez! Bilgi sadece paylaşılarak çoğalır. Lütfen bu yazıyı faydalanacağını düşündüğünüz şahışlara gönderiniz.

Okunması Tavsiye Edilen Diğer Makaleler

- Tasarım Şablonu Nedir?
  http://www.kurumsaljava.com/2008/11/20/tasarim-sablonu-nedir/
- Singleton (Tekillik) Tasarım Şablonu
  http://www.kurumsaljava.com/2008/11/27/singleton-yanlizlik-tasarim-sablonu/
- Data Access Object (DAO) Tasarım Şablonu
  http://www.kurumsaljava.com/2008/12/01/data-access-object-dao-tasarim-sablonu/
- Java'da Interface ve Soyut (Abstract) Sınıf Kullanımı
  http://www.kurumsaljava.com/2008/11/19/javada-interface-ve-soyut-abstract-sinif-kullanimi/
- Iterator Tasarım Şablonu
  http://www.kurumsaljava.com/2008/12/08/iterator-tekrarlayici-tasarim-sablonu

Strategy (Strateji) Tasarım Şablonu

Bir işlemi yapabilmek için birden fazla yöntem (algoritma) mevcut olabilir. Yerine göre bir yöntem seçip, uygulamak için Strategy tasarım şablonu kullanılır. Her yöntem (algoritma) bir sınıf içinde implemente edilir.



Seçtiğim örnekte Bean ismindeki bir sınıfı **iki değişik yöntem** kullanarak persistent[1] hale getireceğiz, yani Bean sınıfından olan bir nesnenin sahip olduğu değişken değerlerini bilgisayarımızın harddiski üzerine yazıp, daha sonra bu değerleri tekrar edinerek Bean nesnesini oluşturacağız. Bean basit bir Java sınıfıdır ve counter ve name isimlerinde iki sınıf değişkenine sahiptir. getXXX() metotları üzerinden sınıf değişkenlerine ulaşılır.

```
package org.javatasarim.pattern.strategy;

import java.io.Serializable;

/**
 * Basit bir java bean sinifi
 *
 * @author Oezcan Acar
 *
 */
public class Bean implements Serializable
{
      private int counter;
      private String name;

      public int getCounter()
      {
            return counter;
      }
      public void setCounter(int counter)
      {
```

[1] Bir verinin bilgibankası ya da başka bir sistem içinde saklanabilir hale getirilmesi işlemi.

```
            this.counter = counter;
      }

      public String getName()
      {
            return name;
      }

      public void setName(String name)
      {
            this.name = name;
      }
}
```

Strategy tasarım şablonunu uygulayabilmemiz için Strategy isminde bir interface sınıfı tanımlamamız gerekiyor. Bu interface sınıfı bünyesinde, alt sınıflar tarafından implemente edilmesi gereken metot yada metotlar bulunur. Kullandığımız örnekte basit bir Java sınıfını harddisk üzerine aktarmak için ***save(Bean bean)*** isminde bir metot tanımlıyoruz. Uygulamak istediğimiz yönteme göre Strategy interface sınıfını implemente eden bir algoritma seçerek, işlemi gerçekleştireceğiz. Strategy interface sınıfı aşağıdaki yapıya sahiptir:

```
package org.javatasarim.pattern.strategy;

/**
 * Strategy üst interface sinifi.
 * Olusturacagimiz algoritmalar
 * bu sinifi implemente ederler.
 *
 * @author Oezcan Acar
 *
 */
public interface Strategy
{
      void save(Bean bean);
}
```

İmplemente etmek istediğimiz ilk yöntem (algoritma) BeanToFileStrategy ismini taşıyor. Bu sınıf bünyesinde, bean nesnesinin sahip olduğu değerleri bir **StringBuilder** nesnesine dönüştürdükten sonra harddisk üzerinde yer alan bean.txt isimli dosyaya aktarıyoruz.

```
package org.javatasarim.pattern.strategy;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;

/**
 * Bean sinifinin sahip olduğu
 * değişken değerlerini bean.txt isimli
 * bir dosyaya yazar.
 *
 * @author Oezcan Acar
 *
 */
```

```
public class BeanToFileStrategy implements Strategy
{

	public void save(Bean bean)
	{
		try
		{
			StringBuilder temp = new StringBuilder();
			temp.append("counter: ")
			    .append(bean.getCounter())
			    .append("\n");

			temp.append("name: ")
			    .append(bean.getName())
			    .append("\n");

			File file = new File("c:/temp/bean.txt");
			if(file.exists())
			{
				file.delete();
				file = new File("c:/temp/bean.txt");
			}

			Writer output = null;
			try
			{
			    output = new BufferedWriter(new FileWriter(file));
			    output.write( temp.toString() );
			}
			finally
			{
				if (output != null) output.close();
			}
			System.out.println("bean.txt olusturuldu.");
		}
		catch (Exception e)
		{
			throw new RuntimeException(e);
		}
	}
}
```

BeanToFileStrategy sınıfı Strategy interface sınıfını implemente ettiği için ***save()*** metoduna sahiptir. Bu metot bünyesinde önce bir **StringBuilder** nesnesi oluşturuyoruz. ***append()*** operasyonu ile bean nesnesinin sahip olduğu değişken isimleri ve değerleri StringBuilder nesnesine eklenir. Bu işlemin ardından File sınıfını kullanarak c:\temp altında bean.txt isminde bir dosya oluşturuyoruz. Eğer bu dosya varsa, silerek, yeniden oluşturulur. **BufferedWriter** sınıfını kullanarak **StringBuilder** bünyesinde bulunan değerleri bu dosyaya aktarıyoruz. Bu işlemin ardından c:\temp dizininde bean.txt isminde bir dosya oluşturulacak ve içeriği aşağıdaki şekilde olacaktır:

```
counter: 10
name: name
```

BeanToFileStrategy sınıfı ile, bir nesnenin sahip olduğu değişken değerlerini harddisk üzerinde bulunan bir dosyaya aktardık. Bu uygulanabilir yöntemlerden bir tanesini teşkil etmektedir. Bean nesnesinin yapısını başka bir yöntem kullanarak ta harddisk üzerinde bulunan bir dosyaya aktarabiliriz. Yeni bir yöntem (algoritma) oluşturma adına

SerializeStrategy isminde ikinci bir sınıf tanımlıyoruz. Bu sınıf **StringBuilder** yerine, bean nesnesini direk zerialize[2] ederek, bean.ser dosyasını oluşturacaktır.

```
package org.javatasarim.pattern.strategy;

import java.io.FileOutputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

/**
 * Bean sinifinin sahip olduğu
 * değişken değerlerini zerialize
 * ederek bean.ser isimli
 * bir dosyaya yazar.
 *
 * @author Oezcan Acar
 *
 */
public class SerializeStrategy implements Strategy
{

	public void save(Bean bean)
	{
		try
		{
			ObjectOutput out = new ObjectOutputStream(
					new FileOutputStream("c:/temp/bean.ser"));
			out.writeObject(bean);
			out.close();
		}
		catch (Exception e)
		{
			throw new RuntimeException(e);
		}
	}
}
```

SerializeStrategy yöntemi kullanılarak harddiske aktarılan bir bean nesnesi için c:\temp dizininde bean.ser isminde bir dosya oluşturulur. Bu dosyayı bir text editörü ile incelediğimiz zaman, aşağıdaki yapıda olduğunu görürüz:


```
¬í__sr_%org.javatasarim.pattern.strategy.Bean_QKy«”žÒ___I_
counterL__namet__Ljava/lang/String;xp___
t__name
```


Binary[3] formatta olan bu dosya bean nesnesinin tüm yapısını ihtiva eder. Bu dosyanın içeriğini kullanarak bean nesnesini tekrar oluşturabiliriz (reconstruction).

Bean nesnesini harddisk üzerinde bulunan bir dosyaya aktarmak için **FileManage**r isminde bir sınıf kullanıyoruz. Bu sınıf aşağıdaki yapıdadır:

```
package org.javatasarim.pattern.strategy;
```

[2] İngilizcesi serialize olan bu yöntem ile bir Java nesnesi yapısı değiştirilmeden binary formatta bir extern dosyaya ya da network üzerinden başka bir servere aktarılır. Örneğin RMI operasyonlarında Java nesnelerinin bir serverden diğer bir servere aktarılabilmeleri için önce zerialize edilmeleri gerekmektedir.
[3] Bakınız: http://en.wikipedia.org/wiki/Binary_data

```java
import java.util.ResourceBundle;

/**
 * FileManager sinifi
 *
 * @author Oezcan Acar
 *
 */
public class FileManager
{
	/**
	 * Bünyesinde bir strategy
	 * nesnesi barindirir.
	 */
	private Strategy strategy;

	/**
	 * Singleton tasarım şablonunu
	 * kullanarak bu siniftan sadece
	 * bir nesnenin olusmasini sagliyoruz.
	 */
	public static final FileManager manager =
		new FileManager();


	/**
	 * Sinif konstrüktörü içinde
	 * strategy.properties dosyasinda
	 * tanimlanmis olan strategy sinifini
	 * yükleyerek, bir strategy nesnesi olusturuyoruz.
	 */
	private FileManager()
	{
		String strategy = ResourceBundle.getBundle(
				"org/javatasarım/pattern/strategy/strategy")
		.getString("strategy");

		try
		{
			setStrategy(((Strategy)Class.forName(strategy)
					.newInstance()));
		}
	    catch (Exception e)
	    {
			throw new RuntimeException(e);
		}
	}

	public static FileManager instance()
	{
		return manager;
	}


	public Strategy getStrategy()
	{
		return strategy;
	}

	public void setStrategy(Strategy strategy)
	{
```

```
            this.strategy = strategy;
      }

      public void saveBean(Bean bean)
      {
            getStrategy().save(bean);
      }
}
```

İlk etapta FileManager bünyesinde, kullanmak istediğimiz strategy nesnesini tutmak için strategy isminde bir sınıf değişkeni tanımlıyoruz. FileManager sınıfını Singleton tasarım şablonuna göre implemente edilmiştir çünkü sahip olduğu tek konstrüktör private olarak deklare edilmiştir. Bu durumda dışardan new operatörü ile bu sınıfın bir nesnesi oluşturulamaz. Static olarak tanımlanan manager ismindeki sınıf değişkeni ile sistemde tek mevcut olan FileManager nesnesine ulaşıyoruz.
Private olarak deklare ettiğimiz sınıf konstrüktöründe yer alan **ResourceBundle** sınıfı dikkatinizi çekmiş olabilir.

```
      private FileManager()
      {
            String strategy = ResourceBundle.getBundle(
                        "org/javatasarım/pattern/strategy/strategy")
            .getString("strategy");

            try
            {
                  setStrategy(((Strategy)Class.forName(strategy)
                              .newInstance()));
            }
          catch (Exception e)
          {
                  throw new RuntimeException(e);
            }
      }
```

Sahip olduğumuz strategy sınıflarını daha esnek bir şekilde program kodunda kullanabilmek için strategy.properties isminde bir dosya tanımlayarak, kullanmak istediğimiz strategy sınıf ismini bu dosyada belirtiyoruz. Strategy.properties sınıfı aşağıdaki yapıya sahiptir:

```
strategy = org.javatasarim.pattern.strategy.BeanToFileStrategy
```

Bu dosya içinde **strategy** isminde bir anahtar ve anahtarın değeri olarak **BeanToFileStrategy** sınıfını tanımlıyoruz. FileManager konstrüktöründe ResourceBundle.getBundle() metodu ile strategy.properties içinde tanımlanmış olan strategy anahtarının değerini edinebiliriz. Akabinde ***Class.forName()*** operasyonu ile new yapmaya gerek kalmadan **BeanToFileStrategy** sınıfından bir nesne oluşturuyoruz. Aynı şekilde sadece strategy.properties içinde başka bir strategy sınıfının ismini vererek, FileManager sınıfı içinde bu sınıftan bir nesne oluşturulmasını sağlayabiliriz. Bir property dosyasını kullanarak, FileManager sınıfını tekrar derlemek (compile) zorunda kalmadan istediğimiz şekilde kullanılan Strategy sınıfını değiştirebiliriz.

Aşağıda yer alan Test programı ile, oluşturduğumuz bir bean nesnesini harddisk üzerinde bir dosyaya aktarıyoruz.

```java
package org.javatasarim.pattern.strategy;

/**
 * Test sinifi
 *
 * @author Oezcan Acar
 *
 */
public class Test
{
	public static void main(String[] args)
	{
		Bean bean = new Bean();
		bean.setCounter(10);
		bean.setName("name");
		FileManager.instance().saveBean(bean);
	}
}
```

Test.main() metodunda görüldüğü gibi oluşturduğumuz bean nesnesini FileManager yardımı ile harddisk üzerinde bir dosyaya transfer etmiş oluyoruz. Test sınıfı, transfer işlemi için kullandığımız yöntemi bile tanımamaktadır. Bu sayede Test sınıfını etkilemeden başka yöntemleri Strategy sınıfı olarak implemente ederek kullanabiliriz.

Strategy tasarım şablonu ne zaman kullanılır?

- Bir işlemi birden fazla yöntem (algoritma) ile implemente etmek için Strategy tasarım şablonu kullanılır. Sistem gereksinimleri doğrultusunda en uygun yöntem seçilerek, işlemin gerçekleştirilmesinde kullanılır.
- Kullanıcı sınıfların, kullanılan yöntemler hakkında bilgi sahibi olmamaları gerektiği durumlarda Strategy tasarım şablonu kullanılır. Kullandığımız örnekte Test sınıfı hangi yöntemin kullanıldığını bilemez. Böylece kullanıcı ile sistem arasında gizli bir duvar örerek, sistemin nasıl çalıştığını kullanıcıdan saklama yeteneğine kavuşmuş oluyoruz.
- Java programlarında switch operatörünün kullanılması kötü bir tasarım yapıldığının işaretidir. Switch ve ona benzer yapılar yerine Strategy tasarım şablonu kullanılarak daha iyi çözümler oluşturulabilir.

İlişkili tasarım şablonları:

- Bir çok kullanıcı sınıf mevcutsa, Strategy nesneleri Flyweight (sineksiklet) olarak implemente edilerek kullanıcı sınıflar tarafından ortaklaşa kullanılmaları sağlanabilir.