# Návrhové vzory II

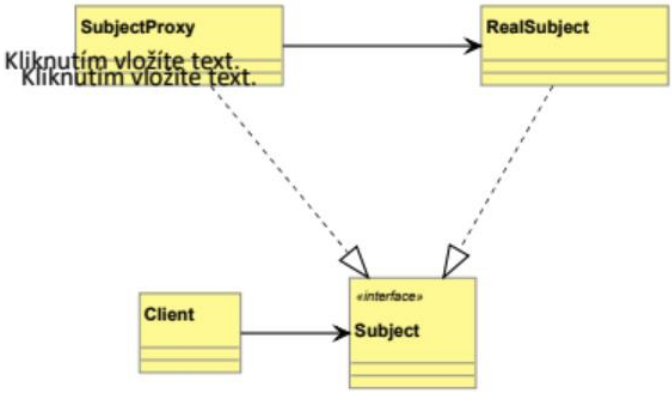
## Návrhové vzory pro skrývání implementace, optimalizaci rozhraní

Joseph Soukup – P4

## Definice

-

## Proxy (zástupce)

- Nahrazuje přístup k zastupované hodnotě
- Implementováno rozhraním nebo abstraktní třídou
- Typy proxy:
  - Remote proxy
  - Virtual proxy
  - Smart reference
  - Protection proxy
  - Copy-on-Write proxy
  - Synchronization proxy
- Příklad: Bankovní šek
  - 

- **REMOTE**
  - Lokální zástupce vzdáleného objektu
- **SMART REFERENCE**
  - Doplnění komunikace mezi objekty
- **COPY-ON-WRITE (COW)**
  - Opožděné kopírování až při modifikaci
- **VIRTUAL**
  - Používá se, když potřebujeme odložit vytvoření objektu
- **PROTECTION**
  - Používá se na skrytí identity
  - Při potřebě ověření přístupových práv
- **SYNCHRONIZATION**
  - Používá se na omezení více lidí k jednomu souboru
  - Synchronizace

```
namespace NV_Prezentace.Proxy
{
    interface Database<K,V>
    {
        V Read(K key);
        void Write(K key, V value);
    }
}
```

```
namespace NV_Prezentace.Proxy
{
    class LoggingDatabase<K, V> : Database<K, V>
    {
        private Database<K, V> databaseDelegate;
        public LoggingDatabase(Database<K, V> database)
        {
            databaseDelegate = database;
        }
        public V Read(K key)
        {
            log("Reading from key: " + key);
            return databaseDelegate.Read(key);
        }

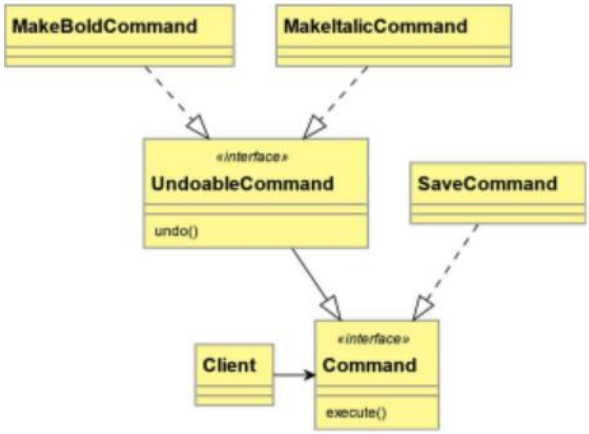
        public void Write(K key, V value)
        {
            log("Writing to key: " + key);
            databaseDelegate.Write(key, value);
        }
        private void log(string message)
        {
            Console.WriteLine(message);
        }
    }
}
```

```
namespace NV_Prezentace.Proxy
{
    class SecureDatabase<K,V> : Database<K,V>
    {
        private Database<K, V> databaseDelegate;
        public SecureDatabase(Database<K,V> database)
        {
            this.databaseDelegate = database;
        }
        public V Read(K key)
        {
            if (CanRead())
                return databaseDelegate.Read(key);
            else
                throw new Exception();
        }
        public void Write(K key, V value)
        {
            if (CanWrite())
                databaseDelegate.Write(key, value);
            else
                throw new Exception();
        }
        private bool CanRead()
        {
            //kontrola přístupu
            return false;
        }
        private bool CanWrite()
        {
            //kontrola přístupu
            return false;
        }
    }
}

namespace NV_Prezentace.Proxy
{
    public class SimpleDatabase<K, V> : Database<K,V>
    {
        private Dictionary<K, V> storage = new Dictionary<K,V>();
        public V Read(K key)
        {
            return storage.GetValueOrDefault(key);
        }
        public void Write(K key, V value)
        {
            storage.Add(key, value);
        }
    }
}
```

# Command (příkaz)

- Používá se za účelem změny funkcí na objekty, se kterými se dá pracovat
- Má připravené metody, ale potřebuje parametry
- Příklad: Objednávka v restauraci




```
#region Command
CommandExample example = new CommandExample();
example.Test();
#endregion
```

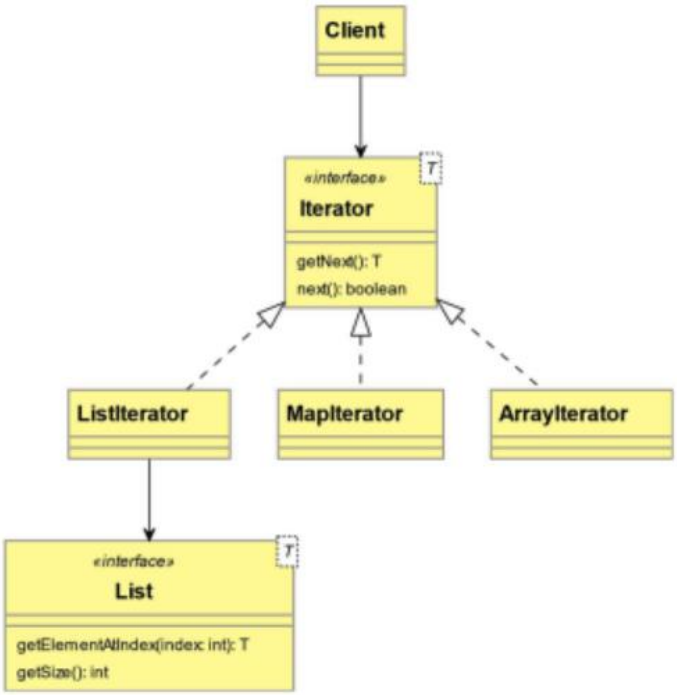
```
namespace NV_Prezentace.Commands
{
    public interface Command
    {
        void Execute();
    }
}
```

```
namespace NV_Prezentace.Commands
{
    public class NewLineCommand : Command
    {
        public void Execute()
        {
            Console.WriteLine();
        }
    }
}
```

```
namespace NV_Prezentace.Commands
{
    class PrintCommand : Command
    {
        private string text;
        public PrintCommand(string rText)
        {
            text = rText;
        }
        public void Execute()
        {
            Console.WriteLine(text);
        }
    }
}
```

# Iterator

- Zprostředkovává přístup k objektům uloženým v kolekci
- Implementace struktury je skryta Externí x Interní
- Příklad: Systematické procházení kolekcemi




```
#region Iterator
int[] array = { 1, 2, 3 };
SimpleIterator<int> iterator = new ForwardArrayIterator<int>(array);
while (iterator.HasNext())
{
    Console.WriteLine(iterator.Next());
}
#endregion
```

```
namespace NV_Prezentace.Iterátor
{
    class ForwardArrayIterator<T> : SimpleIterator<T>
    {
        private T[] array;
        private int index;
        public ForwardArrayIterator(T[] array)
        {
            this.array = array;
            index = -1;
        }
        public bool HasNext()
        {
            return index < array.Length - 1;
        }

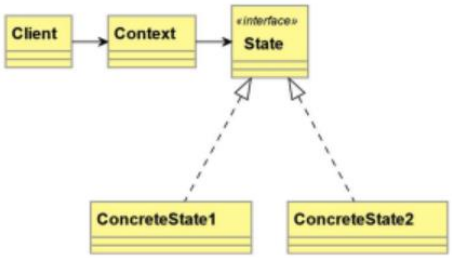
        public T Next()
        {
            index++;
            Debug.Assert(index <= array.Length - 1);
            return array[index];
        }
    }
}
```

```
namespace NV_Prezentace.Iterátor
{
    interface List<T>
    {
        T GetElementAtIndex(int index);
        int GetSize();
    }
}
```

```
namespace NV_Prezentace.Iterátor
{
    interface SimpleIterator<T>
    {
        bool HasNext();
        T Next();
    }
}
```

## State

- Používá se, pokud je veliký rozdíl mezi stavy objektu
- Implementace objektu, který reprezentuje stav pomocí stavových tříd
- Příklad: Prodejní automat, auto s automatickou převodovkou
- Dvě části
    - Stavově závislá
    - Stavově nezávislá

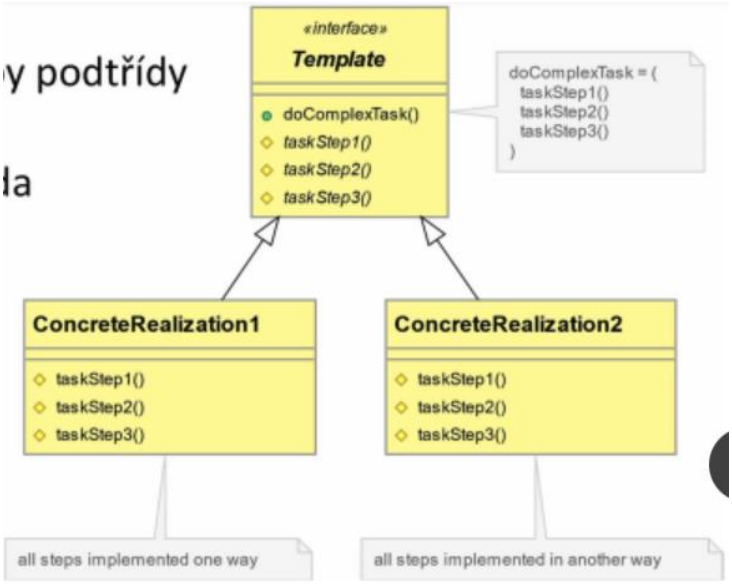



```
#region State
StateExample state = new StateExample();
state.Test();
#endregion
```

```
namespace NV_Prezentace.State
{
    class Context
    {
        private StateOfMind state;
        public Context()
        {
            state = new HappyState();
        }
        public void Express()
        {
            Console.WriteLine("I will tell my feelings.");
            state.Express();
        }
        public void ChangeMood()
        {
            Random rnd = new Random();
            if(rnd.Next(11) < 7)
            {
                Console.WriteLine("Now I will be happy.");
                state = new HappyState();
            }
            else
            {
                Console.WriteLine("Now I will be sad.");
                state = new SadState();
            }
        }
    }
}
```

- ```
  namespace NV_Prezentace.State
  {
      class HappyState : StateOfMind
      {
          public void Express()
          {
              Console.WriteLine("I am happy. :)");
          }
      }
  }
  ```
- ```
  namespace NV_Prezentace.State
  {
      class SadState : StateOfMind
      {
          public void Express()
          {
              Console.WriteLine("I am sad. :(");
          }
      }
  }
  ```
- ```
  namespace NV_Prezentace.State
  {
      interface StateOfMind
      {
          void Express();
      }
  }
  ```
  ```

# Template Method

- Používá se v případě, že chceme aby podtřídy mohly upravovat kroky algoritmů
- Některé funkce definuje až podtřída
- Abstraktní/virtuální metody
- Příklad: Denní rutina zaměstnance
  - 

  - ```
    #region Template
    TemplateExample.TestChild();
    TemplateExample.TestWorker();
    #endregion
    ```
  - ```
    namespace NV_Prezentace.Template
    {
        class Child : Person
        {
            protected override void Relax()
            {
                Console.WriteLine("Playing with other kids.");
            }
            protected override void Sleep()
            {
                Console.WriteLine("Going to bed.");
            }

            protected override void Wake()
            {
                Console.WriteLine("Waking up at 7:00");
            }
            protected override void Work()
            {
                Console.WriteLine("Going to school.");
            }
        }
    }
    ```

```
namespace NV_Prezentace.Template
{
    public abstract class Person
    {
        public void PrintDailyRoutine()
        {
            Wake();
            Work();
            Relax();
            Sleep();
        }
        abstract protected void Work();
        abstract protected void Wake();
        abstract protected void Relax();
        abstract protected void Sleep();
    }
}
```

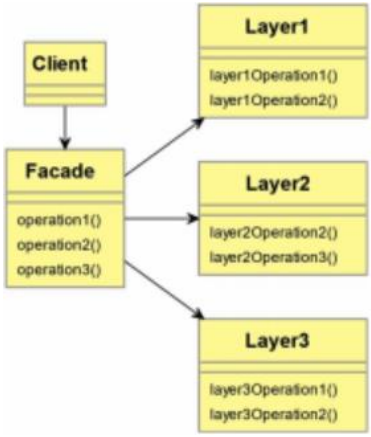
```
namespace NV_Prezentace.Template
{
    public class Worker : Person
    {
        protected override void Relax()
        {
            Console.WriteLine("Watching movies.");
        }

        protected override void Sleep()
        {
            Console.WriteLine("Going to bed.");
        }

        protected override void Wake()
        {
            Console.WriteLine("Waking up at 6:00");
        }
        protected override void Work()
        {
            Console.WriteLine("Going to work. Working...");
        }
    }
}
```

## Facade

- Používá se pro zjednodušení komunikace
- Redukuje počet objektů, se kterými je nutno komunikovat
- Centralizace a snížení duplicitního kódu
- Příklad: Zákaznická podpora

- 

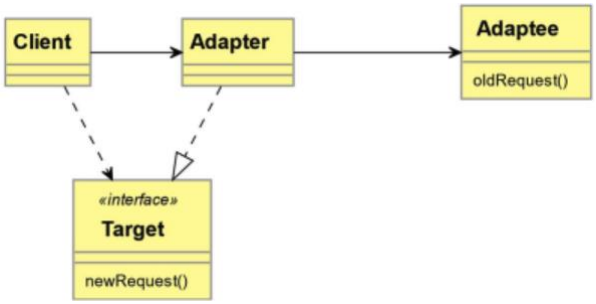
- ```
  #region Facade
  FacadeExample facade = new FacadeExample();
  facade.Test();
  #endregion
  ```
- ```
  namespace NV_Prezentace.FacadeFolder
  {
      class Adder
      {
          public double Add(double a, double b)
          {
              return a + b;
          }
          public double Subtract(double a, double b)
          {
              return a - b;
          }
      }
  }
  ```

```
namespace NV_Prezentace.FacadeFolder
{
    class Facade
    {
        private Adder adder = new Adder();
        private Multiplier multiplier = new Multiplier();
        public double Negative(double a)
        {
            return adder.Subtract(0, a);
        }
        public double Mean(double a, double b)
        {
            return multiplier.Divide(adder.Add(a, b), 2);
        }
        public double Square(double a)
        {
            return multiplier.Multiply(a, a);
        }
    }
}
```

```
namespace NV_Prezentace.FacadeFolder
{
    class Multiplier
    {
        public double Multiply(double a, double b)
        {
            return a * b;
        }
        public double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

## Adapter

- Používáme pro změnu rozhraní objektů
- Překladač vyžaduje kompletní implementaci rozhraní
- Usnadní definici nových tříd a zabezpečí spolupráci již existujících
- Příklad: Cestovní Adaptér




```
#region Adapter
AdapterExample adapter = new AdapterExample();
adapter.Test();
#endregion
```

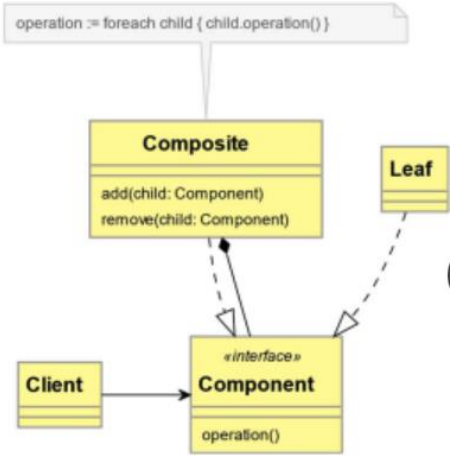
```
namespace NV_Prezentace.AdapterFolder
{
    class Adaptee
    {
        public void OldReqest()
        {
            //akce
        }
    }
}
```

```
namespace NV_Prezentace.AdapterFolder
{
    class Adapter : Target
    {
        private Adaptee adaptee;
        public Adapter()
        {
            adaptee = new Adaptee();
        }
        public void NewRequest()
        {
            //bonus logika kódu
            adaptee.OldReqest();
        }
    }
}
```

```
namespace NV_Prezentace.AdapterFolder
{
    interface Target
    {
        void NewRequest();
    }
}
```

## Composite

- Hierarchická struktura objektů, které mohou ale nemusí obsahovat další objekty
- Všechny třídy implementují jedno rozhraní, ale všechny zaměnitelným způsobem
- Příklad: Složky obsahující soubory nebo podsložky



```
#region Composite
CompositeExample composite = new CompositeExample();
composite.Test();
#endregion
```

```
namespace NV_Prezentace.CompositeFolder
{
    interface Component
    {
        void DoSomething();
    }
}
```

```
namespace NV_Prezentace.CompositeFolder
{
    class Composite : Component
    {
        private List<Component> components;
        public Composite()
        {
            components = new List<Component>();
        }
        public void AddComponent(Component component)
        {
            components.Add(component);
        }
        public void RemoveComponent(Component component)
        {
            components.Remove(component);
        }
        public void DoSomething()
        {
            foreach(Component component in components)
            {
                component.DoSomething();
            }
        }
    }
}
```

```
namespace NV_Prezentace.CompositeFolder
{
    class Leaf : Component
    {
        public void DoSomething()
        {
            Console.WriteLine("This is a call");
        }
    }
}
```