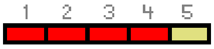
Adapter

dofactory.com/net/adapter-design-pattern

Definition

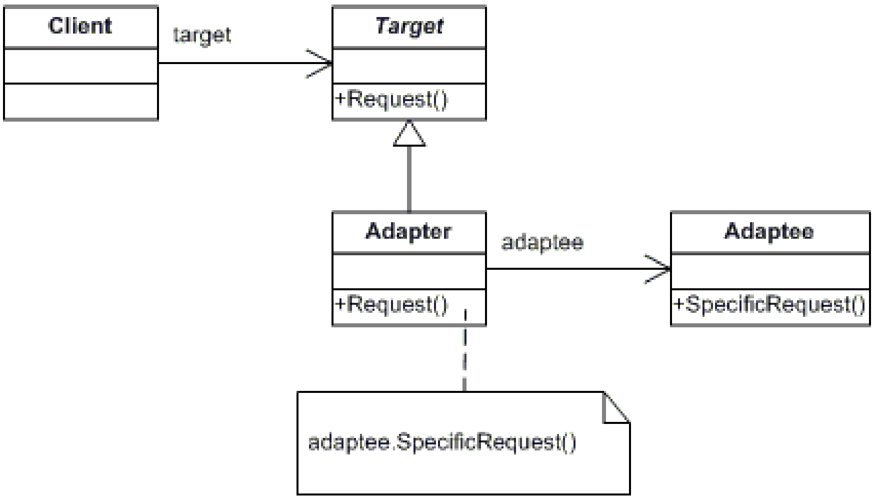
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Frequency of use:



Medium high

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Target (ChemicalCompound)**
  defines the domain-specific interface that Client uses.
- **Adapter (Compound)**
  adapts the interface Adaptee to the Target interface.
- **Adaptee (ChemicalDatabank)**
  defines an existing interface that needs adapting.
- **Client (AdapterApp)**
  collaborates with objects conforming to the Target interface.

Structural code in C#

This structural code demonstrates the Adapter pattern which maps the interface of one class onto another so that they can work together. These incompatible classes may come from different libraries or frameworks.

```
using System;
namespace DoFactory.GangOfFour.Adapter.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Adapter Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Create adapter and place a request
      Target target = new Adapter();
      target.Request();
      // Wait for user
      Console.ReadKey();
    }
  }
  /// <summary>
  /// The 'Target' class
  /// </summary>
  class Target
  {
    public virtual void Request()
    {
      Console.WriteLine("Called Target Request()");
    }
  }
  /// <summary>
  /// The 'Adapter' class
  /// </summary>
  class Adapter : Target
  {
    private Adaptee _adaptee = new Adaptee();
    public override void Request()
    {
      // Possibly do some other work
      //  and then call SpecificRequest
```

```
      _adaptee.SpecificRequest();
    }
  }
  /// <summary>
  /// The 'Adaptee' class
  /// </summary>
  class Adaptee
  {
    public void SpecificRequest()
    {
      Console.WriteLine("Called SpecificRequest()");
    }
  }
}
```

Output

```
Called SpecificRequest()
```

Real-world code in C#

This real-world code demonstrates the use of a legacy chemical databank. Chemical compound objects access the databank through an Adapter interface.

```
using System;
namespace DoFactory.GangOfFour.Adapter.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Adapter Design Pattern.
  /// </summary>
```

```
class MainApp
{
  /// <summary>
  /// Entry point into console application.
  /// </summary>
  static void Main()
  {
    // Non-adapted chemical compound
    Compound unknown = new Compound("Unknown");
    unknown.Display();
    // Adapted chemical compounds
    Compound water = new RichCompound("Water");
    water.Display();
    Compound benzene = new RichCompound("Benzene");
    benzene.Display();
    Compound ethanol = new RichCompound("Ethanol");
    ethanol.Display();
    // Wait for user
    Console.ReadKey();
  }
}
/// <summary>
/// The 'Target' class
/// </summary>
class Compound
{
  protected string _chemical;
  protected float _boilingPoint;
  protected float _meltingPoint;
  protected double _molecularWeight;
  protected string _molecularFormula;
  // Constructor
  public Compound(string chemical)
  {
    this._chemical = chemical;
  }
  public virtual void Display()
  {
    Console.WriteLine("\nCompound: {0} ------ ", _chemical);
  }
}
/// <summary>
/// The 'Adapter' class
/// </summary>
class RichCompound : Compound
{
  private ChemicalDatabank _bank;
  // Constructor
  public RichCompound(string name)
    : base(name)
  {
  }
  public override void Display()
  {
    // The Adaptee
    _bank = new ChemicalDatabank();
    _boilingPoint = _bank.GetCriticalPoint(_chemical, "B");
    _meltingPoint = _bank.GetCriticalPoint(_chemical, "M");
    _molecularWeight = _bank.GetMolecularWeight(_chemical);
    _molecularFormula = _bank.GetMolecularStructure(_chemical);
    base.Display();
    Console.WriteLine(" Formula: {0}", _molecularFormula);
    Console.WriteLine(" Weight : {0}", _molecularWeight);
    Console.WriteLine(" Melting Pt: {0}", _meltingPoint);
    Console.WriteLine(" Boiling Pt: {0}", _boilingPoint);
  }
}
/// <summary>
/// The 'Adaptee' class
/// </summary>
class ChemicalDatabank
{
  // The databank 'legacy API'
  public float GetCriticalPoint(string compound, string point)
  {
    // Melting Point
    if (point == "M")
    {
      switch (compound.ToLower())
      {
        case "water": return 0.0f;
        case "benzene": return 5.5f;
        case "ethanol": return -114.1f;
        default: return 0f;
      }
    }
    // Boiling Point
  else
  {
    switch (compound.ToLower())
    {
      case "water": return 100.0f;
      case "benzene": return 80.1f;
      case "ethanol": return 78.3f;
      default: return 0f;
    }
  }
}
public string GetMolecularStructure(string compound)
{
  switch (compound.ToLower())
  {
    case "water": return "H20";
    case "benzene": return "C6H6";
    case "ethanol": return "C2H5OH";
    default: return "";
  }
}
public double GetMolecularWeight(string compound)
{
  switch (compound.ToLower())
  {
    case "water": return 18.015;
    case "benzene": return 78.1134;
    case "ethanol": return 46.0688;
    default: return 0d;
```

```
   }
  }
 }
}
```

Output

```
Compound: Unknown ------

Compound: Water ------
 Formula: H20
 Weight : 18.015
 Melting Pt: 0
 Boiling Pt: 100

Compound: Benzene ------
 Formula: C6H6
 Weight : 78.1134
 Melting Pt: 5.5
 Boiling Pt: 80.1

Compound: Alcohol ------
 Formula: C2H6O2
 Weight : 46.0688
 Melting Pt: -114.1
 Boiling Pt: 78.3
```

.NET Optimized code in C#

The .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

You can find an example on our Singleton pattern page.

All other patterns (and much more) are available in our **.NET Design Pattern Framework 4.5.**

Not only does the **.NET Design Pattern Framework 4.5** cover GOF and Enterprise patterns, it also includes .NET pattern architectures that reduce the code you need to write by up to 75%. This unique package will change your .NET lifestyle -- for only $79. Here's what is included:

- 69 gang-of-four pattern projects
- 46 head-first pattern projects
- Fowler's enterprise patterns
- Multi-tier patterns
- Convention over configuration
- Active Record and CQRS patterns
- Repository and Unit-of-Work patterns
- MVC, MVP, & MVVM patterns
- REST patterns with Web API
- Spark^{TM} Rapid App Dev (RAD) platform!
- Art Shop MVC Reference Application
- 100% pure source code

Discover the secrets of expert .NET developers

Creational Patterns

Structural Patterns

Behavioral Patterns

Reference Guides

Our Products