|  |
| --- |
| University of Michigan - dearborn |
| CIS 476 Project 2 |
| By Scott Smereka |
|  |
|  |
|  |

|  |
| --- |
|  |

Table of Contents

[Problem 1 2](#_Toc289708814)

[Research 2](#_Toc289708815)

[UML Class Diagram 2](#_Toc289708816)

[Example: Problem 1 Handler 2](#_Toc289708817)

[Example: Problem 1 Client 3](#_Toc289708818)

[UML Diagram 4](#_Toc289708819)

[Execution Scenarios 4](#_Toc289708820)

[Scenario 1 & 6 4](#_Toc289708821)

[Scenario 2 5](#_Toc289708822)

[Scenario 3 5](#_Toc289708823)

[Scenario 4 5](#_Toc289708824)

[Scenario 5 5](#_Toc289708825)

[Source Code 5](#_Toc289708826)

[MainForm.cs 5](#_Toc289708827)

[Handler.cs 7](#_Toc289708828)

[Problem 2 8](#_Toc289708829)

[Research Decorator Pattern 8](#_Toc289708830)

[UML Class Diagram 8](#_Toc289708831)

[Example: Problem 2 Decorator 9](#_Toc289708832)

[Example: Problem 2 Main Form 9](#_Toc289708833)

[UML Diagram 10](#_Toc289708834)

[Execution Scenarios 11](#_Toc289708835)

[Scenario 1 & 6 11](#_Toc289708836)

[Scenario 2 11](#_Toc289708837)

[Scenario 3 11](#_Toc289708838)

[Scenario 4 12](#_Toc289708839)

[Scenario 5 12](#_Toc289708840)

[Source Code 12](#_Toc289708841)

[Decorator.cs 12](#_Toc289708842)

[Handler.cs 13](#_Toc289708843)

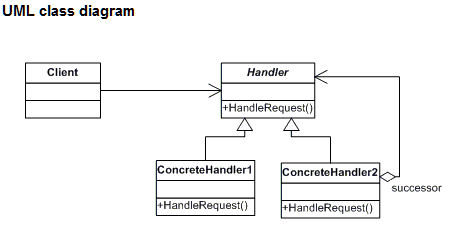
[MainForm.cs 14](#_Toc289708844)

# Problem 1

## Research

The Chain of Responsibility (COR) pattern allows a developer to handle requests, often with a single call, in a dynamic way. The key to the COR pattern is to avoid coupling the sender (or client) to the receiver (Handler). This allows the client to assume the handler will get the job done, or take whatever action is necessary; the client does not care how the handler handles the request. Handlers are chained together and the request is sent to each handler in the chain until a request is handled. Any number of handlers can be chained together, and the actual chain itself can be altered. Handlers may or may not be responsible for handling an event such as “a request is not handled or supported”, but this is up to the implementation of the pattern.

### UML Class Diagram



The participants in the COR pattern consist of an abstract handler class, concrete handlers, and a client. Each concrete handler is inherited from the abstract handler class and performs it own implementation of the “HandleRequest()” operation. In the example below you can see both the abstract class “Handler” and the concrete Class “USDHandler.” The method “Process(String str)” has been overridden with its own implementation. Also note that the function also checks to see if it should handle the request and if it can’t forwards the request to its defined successor.

### Example: Problem 1 Handler

abstract class Handler

{

protected Handler successor;

public Handler Successor

{

get { return this.successor; }

set { this.successor = value; }

}

public abstract Double Process(String str);

}

class USDHandler : Handler

{

public override Double Process(String conversion)

{

Double convFactor = 1.4222;

String[] str = conversion.Split(' ');

if (str.Length == 4)

{

if ((str[1].ToLower() + str[2].ToLower()

+ str[3].ToLower()) == "eurotousd")

return Math.Round(Double.Parse(str[0]) \* convFactor, 2);

}

return Successor.Process(conversion);

}

}

The client is responsible for setting up the chain of handlers before making a request. The order is also determined and of course the successors can be changed at any time. Check out the example below. Also take note that an error handler was added to the end of the chain, this handler will throw an exception. By using a try-catch block we can then tell the user that there was not a handler available for their requested operation.

### Example: Problem 1 Client

public partial class MainForm : Form

{

Boolean evaluated = false;

USDHandler USD = new USDHandler();

CADHandler CAD = new CADHandler();

AUDHandler AUD = new AUDHandler();

ErrorHandler ERH = new ErrorHandler();

private void ConvertBtn\_Click(object sender, EventArgs e)

{

USD.Successor = CAD; //Setup successor chain

CAD.Successor = AUD;

AUD.Successor = ERH; //Handle invalid input with error handler

try {

OutputTxt.Text = "$" + USD.Process(inputTxt.Text); //Try to process data

}

catch{

MessageBox.Show("Error: The entered conversion is not valid");

}

finally{

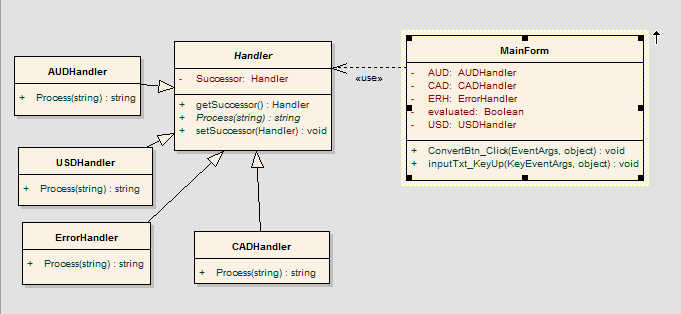
evaluated = true; //Mark the data as being evaluated

ConvertBtn.Enabled = false; //Diable ability to run convert on old data

}

}

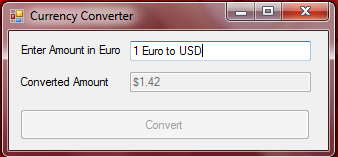
## UML Diagram



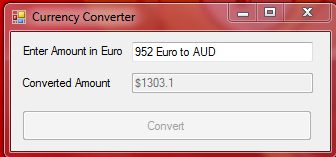
## Execution Scenarios

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Scenario | Input | Expected Output | Actual Output |
| 1 | Convert to USD | 1 Euro to USD | $1.42 | $1.42 |
| 2 | Convert to AUD | 952 Euro to AUD | $1303.1 | $1303.1 |
| 3 | Convert to CAD | -55 Euro to CAD | $-75.33 | $-75.33 |
| 4 | Invalid Input | 1 USD to Euro | Message Box: Error the entered conversion is not valid | Message Box: Error the entered conversion is not valid |
| 5 | No input | Nothing | Should not be able to click the convert button, pressing enter should result in nothing happening | Convert button cannot be clicked and pressing enter yields no results. |
| 6 | Enter same input twice | 1 Euro to USD, click convert, click convert again. | Same as before, should not be able to click convert a second time. | Same as before, was not able to click convert a second time. |

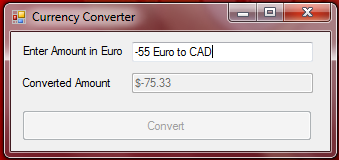
### Scenario 1 & 6



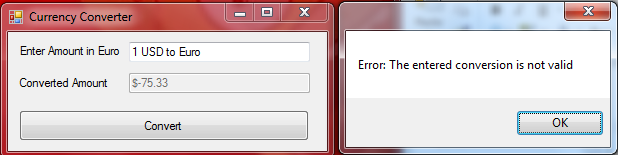
### Scenario 2



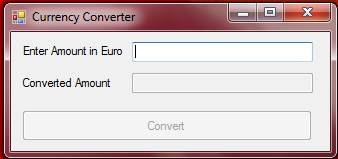
### Scenario 3



### Scenario 4



### Scenario 5



## Source Code

### MainForm.cs

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

namespace CurrencyConverter

{

public partial class MainForm : Form

{

Boolean evaluated = false;

USDHandler USD = new USDHandler();

CADHandler CAD = new CADHandler();

AUDHandler AUD = new AUDHandler();

ErrorHandler ERH = new ErrorHandler();

public MainForm()

{

InitializeComponent();

}

private void ConvertBtn\_Click(object sender, EventArgs e)

{

USD.Successor = CAD; //Setup successor chain

CAD.Successor = AUD;

AUD.Successor = ERH; //Handle invalid input with error handler

try

{

OutputTxt.Text = "$" + USD.Process(inputTxt.Text); //Try to process data

}

catch

{

MessageBox.Show("Error: The entered conversion is not valid");

//Show error

}

finally

{

evaluated = true; //Mark the data as being evaluated

ConvertBtn.Enabled = false; //Diable ability to run convert on old data

}

}

private void inputTxt\_KeyUp(object sender, KeyEventArgs e)

{//Pre: none

//Post: Enable or diable convert process,

// Run input if enter is pressed (and if ok to run it)

if (!e.KeyCode.Equals(Keys.Enter)) //If new input to process

evaluated = false;

//Mark data as new data to evaluate

if ((inputTxt.Text != "") && (!evaluated))

//If not blank, and not a previously processed

ConvertBtn.Enabled = true; //Enable the button

else

ConvertBtn.Enabled = false;

//Diable button if blank or already processed

//If entering an input, and currently allowed to

if (e.KeyCode.Equals(Keys.Enter) && ConvertBtn.Enabled == true)

ConvertBtn\_Click(sender, e); //Run convert

}

}

}

### Handler.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace CurrencyConverter

{

abstract class Handler

{

protected Handler successor;

public Handler Successor

{

get { return this.successor; }

set { this.successor = value; }

}

public abstract Double Process(String str);

}

class USDHandler : Handler

{

public override Double Process(String conversion)

{

Double convFactor = 1.4222;

String[] str = conversion.Split(' ');

if (str.Length == 4)

{

if ((str[1].ToLower() + str[2].ToLower()

+ str[3].ToLower()) == "eurotousd")

return Math.Round(Double.Parse(str[0]) \* convFactor, 2);

}

return Successor.Process(conversion);

}

}

class CADHandler : Handler

{

public override Double Process(String conversion)

{

Double convFactor = 1.3697;

String[] str = conversion.Split(' ');

if (str.Length == 4)

{

if ((str[1].ToLower() + str[2].ToLower()

+ str[3].ToLower()) == "eurotocad")

return Math.Round(Double.Parse(str[0]) \* convFactor, 2);

}

return Successor.Process(conversion);

}

}

class AUDHandler : Handler

{

public override Double Process(String conversion)

{

Double convFactor = 1.3688;

String[] str = conversion.Split(' ');

if (str.Length == 4)

{

if ((str[1].ToLower() + str[2].ToLower()

+ str[3].ToLower()) == "eurotoaud")

return Math.Round(Double.Parse(str[0]) \* convFactor, 2);

}

return Successor.Process(conversion);

}

}

class ErrorHandler : Handler

{

public override double Process(string str)

{

throw new NotImplementedException();

}

}

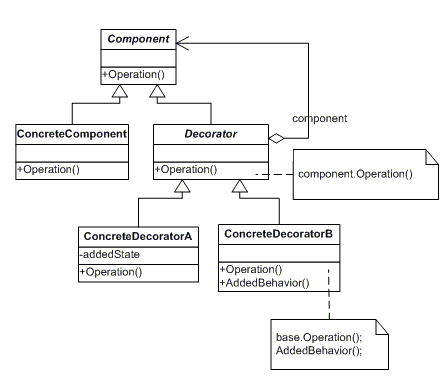
}

# Problem 2

## Research Decorator Pattern

The Decorator pattern is used to attach additional responsibilities to objects dynamically. This provides the developer with an alternative to sub classing extended functionality (dofactory.com). Basically it will act as a wrapper for the concrete component classes, adding additional tasks to the component.

### UML Class Diagram



The component is an abstract class that everything will inherit from, in our code example from problem one this was the Handler class. The concrete component inherits from the component class and will possibly override the abstract method “Operation.” In our previous example a concrete component might be “USDHandler.” A decorator is an abstract class that inherits from the component abstract class and therefore must conform to the components interface. Finally a concrete decorator inherits from the decorator class and adds responsibilities to the component.

### Example: Problem 2 Decorator

abstract class Decorator : Handler

{

protected Handler handler;

public Handler Handler

{

get { return this.handler; }

set { this.handler = value; }

}

public override String Process(String str)

{

if (handler != null)

return handler.Process(str);

return "";

}

}

class RoundDecorator : Decorator

{

public override String Process(String str)

return (Math.Round(Double.Parse(base.Process(str)), 2)).ToString();

}

Taking a look at the example above we can see the “RoundDecorator” is a concrete decorator and rounds the result of the concrete component (in this case the concrete Handler). You can also see that the decorator inherits from the component class (or handler class) and implements the same methods. Taking a look at another example we can see how the client adds this functionality to the handlers.

### Example: Problem 2 Main Form

public partial class MainForm : Form

{

Boolean evaluated = false;

USDHandler USD = new USDHandler();

CADHandler CAD = new CADHandler();

AUDHandler AUD = new AUDHandler();

ErrorHandler ERH = new ErrorHandler();

RoundDecorator Round = new RoundDecorator();

ExpNotationDecorator ExpNote = new ExpNotationDecorator();

CurrencyNameDecorator CurrName = new CurrencyNameDecorator();

private void ConvertBtn\_Click(object sender, EventArgs e)

{

USD.Successor = CAD;

CAD.Successor = AUD;

AUD.Successor = ERH;

Round.Handler = USD; //Go through handler chain, then round

ExpNote.Handler = Round; //Then Convert to exp notation

CurrName.Handler = ExpNote; //Finally Add the currency name to the end

Try

{

OutputTxt.Text = "$" + CurrName.Process(inputTxt.Text);

}

catch {

MessageBox.Show("Error: The entered conversion is not valid");

}

finally {

evaluated = true; //Mark the data as being evaluated

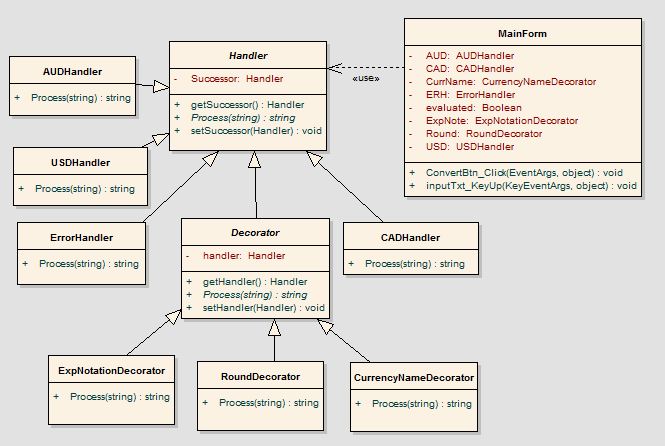
ConvertBtn.Enabled = false; //Diable ability to run convert on old data

}

}

After the chain of handlers is created you can then assign additional functionality to the handler using decorators. A decorator is pointed to the first handler in the chain and then the decorator operation is called. Take note that we can add multiple decorators which will run in the order of closest to the handler first. In our example “Round” will point to the “USDHandler”, the “ExpNote” points to the “Round” decorator, and “CurrName” points to the “ExpNote” decorator. When running the process we run the farthest from the handler decorator “CurrName.Process(“Some string”);”. This will then get the value returned from the “USDHandler”, round it to two decimal places (“Round”), put the result into exponential notation (“ExpNote”), and finally add the name of the currency(“CurrName”).

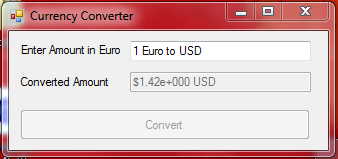
## UML Diagram



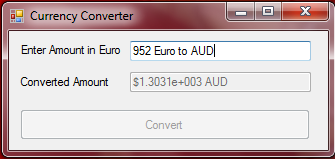
## Execution Scenarios

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Scenario | Input | Expected Output | Actual Output |
| 1 | Convert to USD | 1 Euro to USD | $1.42e+000 USD | $1.42e+000 USD |
| 2 | Convert to AUD | 952 Euro to AUD | $1.3031e+003 AUD | $1.3031e+003 AUD |
| 3 | Convert to CAD | -55 Euro to CAD | $-7.533e+001 CAD | $-7.533e+001 CAD |
| 4 | Invalid Input | 1 USD to Euro | Message Box: Error the entered conversion is not valid | Message Box: Error the entered conversion is not valid |
| 5 | No input | Nothing | Should not be able to click the convert button, pressing enter should result in nothing happening | Convert button cannot be clicked and pressing enter yields no results. |
| 6 | Enter same input twice | 1 Euro to USD, click convert, click convert again. | Same as before, should not be able to click convert a second time. | Same as before, was not able to click convert a second time. |

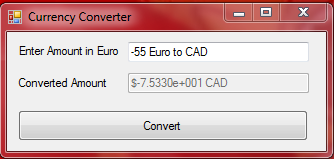
### Scenario 1 & 6



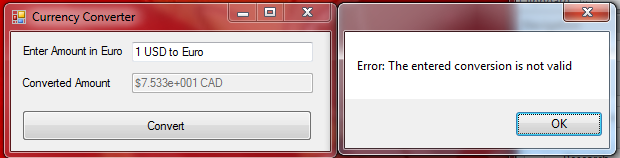
### Scenario 2



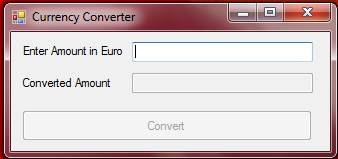
### Scenario 3



### Scenario 4



### Scenario 5



## Source Code

### Decorator.cs

/\* Project 2 CIS 476 Design Patterns

\* By Scott Smereka

\* Problem 2

\*/

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace CurrencyConverter

{

abstract class Decorator : Handler

{

protected Handler handler;

public Handler Handler

{

get { return this.handler; }

set { this.handler = value; }

}

public override String Process(String str)

{

if (handler != null)

return handler.Process(str);

return "";

}

}

class RoundDecorator : Decorator

{

public override String Process(String str)

{

return (Math.Round(Double.Parse(base.Process(str)), 2)).ToString();;

}

}

class ExpNotationDecorator : Decorator

{

public override String Process(String str)

{

return Double.Parse(base.Process(str)).ToString("e"

+(base.Process(str).Length-2).ToString());

}

}

class CurrencyNameDecorator : Decorator

{

public override string Process(string str)

{

String[] input = str.Split(' ');

return base.Process(str) + " " + input[input.Length - 1].ToUpper();

}

}

}

### Handler.cs

/\* Project 2 CIS 476 Design Patterns

\* By Scott Smereka

\* Problem 2

\*/

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace CurrencyConverter

{

abstract class Handler

{

protected Handler successor;

public Handler Successor

{

get { return this.successor; }

set { this.successor = value; }

}

public abstract String Process(String str);

}

class USDHandler : Handler

{

public override String Process(String conversion)

{

Double convFactor = 1.4222;

String[] str = conversion.Split(' ');

if (str.Length == 4)

{

if ((str[1].ToLower() + str[2].ToLower()

+ str[3].ToLower()) == "eurotousd")

return (Double.Parse(str[0]) \* convFactor).ToString();

}

return Successor.Process(conversion);

}

}

class CADHandler : Handler

{

public override String Process(String conversion)

{

Double convFactor = 1.3697;

String[] str = conversion.Split(' ');

if (str.Length == 4)

{

if ((str[1].ToLower() + str[2].ToLower()

+ str[3].ToLower()) == "eurotocad")

return (Double.Parse(str[0]) \* convFactor).ToString();

}

return Successor.Process(conversion);

}

}

class AUDHandler : Handler

{

public override String Process(String conversion)

{

Double convFactor = 1.3688;

String[] str = conversion.Split(' ');

if (str.Length == 4)

{

if ((str[1].ToLower() + str[2].ToLower()

+ str[3].ToLower()) == "eurotoaud")

return (Double.Parse(str[0]) \* convFactor).ToString();

}

return Successor.Process(conversion);

}

}

class ErrorHandler : Handler

{

public override String Process(string str)

{

throw new NotImplementedException();

}

}

}

### MainForm.cs

/\* Project 2 CIS 476 Design Patterns

\* By Scott Smereka

\* Problem 2

\*/

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

namespace CurrencyConverter

{

public partial class MainForm : Form

{

Boolean evaluated = false;

USDHandler USD = new USDHandler();

CADHandler CAD = new CADHandler();

AUDHandler AUD = new AUDHandler();

ErrorHandler ERH = new ErrorHandler();

RoundDecorator Round = new RoundDecorator();

ExpNotationDecorator ExpNote = new ExpNotationDecorator();

CurrencyNameDecorator CurrName = new CurrencyNameDecorator();

public MainForm()

{

InitializeComponent();

}

private void ConvertBtn\_Click(object sender, EventArgs e)

{

//Setup handler successor chain (Order is not important,

//but error handler must be last)

USD.Successor = CAD;

CAD.Successor = AUD;

AUD.Successor = ERH;

//Handle invalid input with error handler as last in chain

//Setup decorator chain (in specific order)

Round.Handler = USD; //Go through handler chain,

//then Round to two decimals

ExpNote.Handler = Round; //Then Convert to exp notation

CurrName.Handler = ExpNote; //Finally Add the currency name to the end

try

{

OutputTxt.Text = "$" + CurrName.Process(inputTxt.Text);

//Try to process data

}

catch

{

MessageBox.Show("Error: The entered conversion is not valid");

//Show error

}

finally

{

evaluated = true; //Mark the data as being evaluated

ConvertBtn.Enabled = false; //Diable ability to run convert on old data

}

}

private void inputTxt\_KeyUp(object sender, KeyEventArgs e)

{//Pre: none

//Post: Enable or diable convert process,

// Run input if enter is pressed (and if ok to run it)

if (!e.KeyCode.Equals(Keys.Enter)) //If new input to process

evaluated = false; //Mark data as new data to evaluate

//If not blank, and not a previously processed

if ((inputTxt.Text != "") && (!evaluated)) ConvertBtn.Enabled = true; //Enable the button

else

ConvertBtn.Enabled = false;

//Diable button if blank or already processed

//If entering an input, and currently allowed to

if (e.KeyCode.Equals(Keys.Enter) && ConvertBtn.Enabled == true)

ConvertBtn\_Click(sender, e); //Run convert

}

}

}