

Level 200 – mixins

Hands on Labs

Copyright notice: ©2011 rubicon informationstechnologie gmbh  all rights reserved

Table of Contents

[1 Lab 1: First steps with mixins 1](#_Toc283883529)

[1.1 Goal 1](#_Toc283883530)

[1.2 Theory – Why mixins 1](#_Toc283883531)

[1.3 The Sample 1](#_Toc283883532)

[1.4 Exercise 1 – OOP Solution without mixins 2](#_Toc283883533)

[1.4.1 Task 1 – The Company Hierarchy 2](#_Toc283883534)

[1.4.2 Questions 4](#_Toc283883535)

[1.4.3 Task 2 – Security representative 4](#_Toc283883536)

[1.4.4 Questions 5](#_Toc283883537)

[1.5 Exercise 2 – First Mixin Implementation 5](#_Toc283883538)

[1.5.1 Task 5](#_Toc283883539)

[1.5.2 Task- ten minutes mixin 5](#_Toc283883540)

[1.5.3 Questions 7](#_Toc283883541)

[1.5.4 Exercise 2 – Parameter 7](#_Toc283883542)

[1.5.5 Questions 8](#_Toc283883543)

[1.6 Exercise 3 – SecurityContext 8](#_Toc283883544)

[1.7 Exercise 3 – A use case for „use“ 8](#_Toc283883545)

[1.7.1 Questions 9](#_Toc283883546)

[2 Lab 2 Composition Pattern 10](#_Toc283883547)

[2.1 Exercise 1 – Adding CompositionPattern 10](#_Toc283883548)

[2.1.1 Task Add 10](#_Toc283883549)

[2.1.2 Task Discuss 12](#_Toc283883550)

[2.2 Task 2 – Add 12](#_Toc283883551)

[3 Lab Summary 14](#_Toc283883552)

[4 Common Practices 14](#_Toc283883553)

[4.1 Getting re-motion 14](#_Toc283883554)

[4.2 Adding references 15](#_Toc283883555)

# Lab 1: First steps with mixins

## Goal

Estimated Time: 30 Minutes

Goal of this HOL is to outline the advantages of mixins by programming examples. All exercises are created to be solved step by step. At the end of each exercise, you should be able to finish it with an executable result.

At the end of an exercise, in some cases you will find a section with questions. We suggest that you answer the questions as well as possible. Some answers to some questions might be straight forward, if you are an experienced developer. Other questions could be tricky. In case you are doing this exercise with other software developers, we encourage you to talk about your ideas and your proposed solutions.

After this exercise, you will be able to understand the advantages of mixins. You will be able to decide when a mixin is appropriate and when not and you are ready to implement mixins in common scenarios.

Please also have a look at the further readings section in the end of this HOL if you want to dig deeper

Topics covered:

|  |
| --- |
| * Basics: What are mixins * Developing your first mixin * Understanding the difference between “use” and “extends” implementations * Parameters |

**Important:**

For this HOL you must have Visual Studio 2010 installed on your working PC. We recommend installing additional third party tools such as Jetbrains Resharper (<http://www.jetbrains.com/resharper/>) or Redgate Reflector (<http://reflector.red-gate.com/Download.aspx>) or equivalent tools on your working PC too. This will help you to understand some samples better and you are able to dig deeper.

If you have no idea how to get re-motion, in the appendix “Getting re-motion“ there is a guide of how to get a version of re-motion. All samples were tested with version 1.13.87.

## Theory – Why mixins

It is essential that you understand the purpose of mixins before you implement your first mixins. The discussion on why mixins are a powerful extension to .NET development is out of scope for this document.

If you are completely new to mixins, it is recommended that you read one or two basic articles on mixins before you get started.

## The Sample

In this sample we want to develop a task planning application to compare solution created with pure OOP and mixin solutions.

There is no complete set of requirements in the beginning of this sample. Additional requirements will be added with each new exercise. Some developers might claim that this makes it more difficult to get an optimized solution and that it would be better to get all requirements in advance.

This is a valid argument, but on the other hand changing requirements are realistic and they will also highlight some advantages of mixins in terms of flexibility.

All we know in the beginning is that the company has different tasks for managers and normal employees. These tasks shall be written to a console.

Although it might not be the best practice, in this sample we keep all approaches and exercises in one solution. This makes it easier to compare the approaches.

C:\Users\stefan.papp\Desktop\Abbildung1.tif

Illustration 1: Object Tree

## Exercise 1 – OOP Solution without mixins

### Task 1 – The Company Hierarchy

In the first exercise we want to solve this scenario without mixins.

1. Start Visual Studio 2010
2. Add a new class library and a new console application to the new solution

Use the following settings

Solution name: HOLMixins

Location: C:\HOL\HOLMixins

Class library Project Name: OOPDomain

Console Application Project Name: OOPConsoleApplication

1. Rename class.cs, call it oopdomainsample.cs.

using System;

using System.Collections.Generic;

namespace OOPDomain

{

public class Person

{

protected Person() { }

protected Person(string firstName, string lastName)

{

FirstName = firstName;

LastName = lastName;

}

public string FirstName { get; set; }

public string LastName { get; set; }

}

public class CoWorker : Person

{

public CoWorker(string firstName, string lastName) : base(firstName, lastName)

{

}

public virtual void FirstTask() { }

}

public class Employee : CoWorker

{

public Manager Manager { get; set; }

public Employee(string firstName, string lastName) : base(firstName, lastName) { }

public override void FirstTask()

{

Console.WriteLine(LastName + " has to look for todays agenda");

}

}

public class Manager : CoWorker

{

public Manager(string firstName, string lastName) : base(firstName, lastName) { }

public override void FirstTask()

{

Console.WriteLine(LastName + " has to give orders");

}

}

}

You might want to put the classes into separated files, but as this is just a demo implementation, this is not required.

1. Replace the code of the default program.cs with the following code.

using System;

using System.Collections.Generic;

using OOPDomain;

namespace OOPConsoleApplication

{

class Program

{

static void Main()

{

var c = CreateCoWorkers();

foreach (CoWorker cw in c)

{

FirstTasksInTheMorning(cw);

}

Console.ReadKey();

}

private static void FirstTasksInTheMorning(CoWorker cw)

{

cw.FirstTask();

}

private static IEnumerable<CoWorker> CreateCoWorkers()

{

List<CoWorker> coWorkers = new List<CoWorker>();

Manager manager = new Manager("William", "Adama");

Employee employee = new Employee("Kara", "Thrace");

Employee employee2 = new Employee("Galen", "Tyrol");

employee.Manager = manager;

employee2.Manager = manager;

coWorkers.Add(manager);

coWorkers.Add(employee);

coWorkers.Add(employee2);

return coWorkers;

}

}

}

1. Add OOPDomain as a reference to OOPConsoleApplication
2. Set OOPConsoleApplication as startup project if necessary and run the application

### Questions

* What does this code? Is this implementation a good approach?
* Let’s think about a possible change! Your customer asks you to implement another scenario. There are some security concerns and it happened that some stuff broke as nobody inspected the machines as very first task in the beginning of a work shift. There is a security representative, who should do that. How would you implement that?
* You might have come to the conclusion that there are two solutions: Composition and Multiple Inheritance. Apart from the fact that Multiple Inheritance is no possible solution, look up the term Diamond Problem.
* How would you solve this additional requirement with Composition?

### Task 2 – Security representative

1. We add now the following lines to the the oopdomainsample.cs file

public abstract class Role { }

public class SecurityRepresentative : Role

{

public void GetWorkPackage(string personName)

{

Console.WriteLine(personName + " has to inspect the flight deck for frakking cylon spies");

}

}

1. We have to adapt the CoWorker code

public CoWorker(string firstName, string lastName)

: base(firstName, lastName)

{

Roles = new List<Role>();

}

public List<Role> Roles { get; private set; }

1. In the CreateCoWorkers() method of the ConsoleApplication, we add

employee2.Roles.Add(new SecurityRepresentative());

1. We add to FirstTasksInTheMorning method

Role r = cw.Roles.Find(role => role.GetType() == typeof(SecurityRepresentative));

if (r != null)

{

((SecurityRepresentative)r).GetWorkPackage(cw.LastName);

}

### Questions

* Now we have added the role security representative. Imagine that the customer comes again. He might want other new business rules. How would you solve that with OOP?
* The way this sample is done, it is probably not the best way to solve it. Can you think about other approaches?

## Exercise 2 – First Mixin Implementation

### Task

Think about

“Composite Oriented Programming builds on some principles that are not addressed by Object Oriented Programming at all.

* Behavior depends on Context
* Decoupling is a virtue
* Business Rules matters more.
* Classes are dead, long live **interfaces**.”

### Task- ten minutes mixin

1. Add two new class library project and a console application project to your solution.

* Class library project Name: MixinDomain
* Class library project Name: MixinDomain.Interfaces
* Console Application Project Name: MixinConsoleApplication

1. Add the following references to the domain project and console app project

* remotion.dll
* remotion.interfaces.dll

You might want to read the section “Adding references” in case you are not sure where to put the binary assemblies.

1. Add a reference from the domain and interfaces projects to the console app.
2. Add a reference from the interface project to the domain project
3. Rename class1.cs from Interfaces to Interfaces.cs and replace code with

namespace MixinDomain.Interfaces

{

public interface IPerson

{

string FirstName { get; set; }

string LastName { get; set; }

}

}

1. Add a file person.cs to the Domain project and use the following source code

using MixinDomain.Interfaces;

namespace MixinDomain

{

public class Person : IPerson

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

}

1. Add a file CoWorkerMixin.cs the domain and use the following source code
2. Add the interfaces

public interface IPerson

{

string FirstName { get; set; }

string LastName { get; set; }

}

public interface IEmployee

{

void GetWorkPackages();

IPerson Manager { get; set; }

}

1. Add the following code to the console

using System;

using System.Collections.Generic;

using MixinDomain;

using MixinDomain.Interfaces;

using Remotion.Mixins;

using Remotion.Reflection;

namespace MixinConsoleApplication

{

class Program

{

private static readonly List<Person> employees = new List<Person>();

private static readonly List<Person> managers = new List<Person>();

static void Main()

{

CreateCoWorkers();

foreach (Person p in employees)

{

((IEmployee)p).GetWorkPackages();

}

foreach (Person p in managers)

{

((IManager)p).GetWorkPackages();

}

Console.ReadKey();

}

private static void CreateCoWorkers()

{

var adama = ObjectFactory.Create<Person>(ParamList.Create());

adama.FirstName = "William";

adama.LastName = "Adama";

managers.Add(adama);

var employee = CreateEmployee("Galen", "Tyrol", adama);

var employee2 = CreateEmployee("Kara", "Thrace", adama);

}

private static Person CreateEmployee(string firstName, string lastName, IPerson manager)

{

var employee = ObjectFactory.Create<Person>(ParamList.Create());

employee.FirstName = firstName;

employee.LastName = lastName;

((IEmployee)employee).Manager = manager;

employees.Add(employee);

return employee;

}

}

}

1. Debug and have a look at the result.

Within a very short time, you have learned that is very easy to implement a basic mixin within your solution. In the upcoming tasks, we want to dig deeper and explore some scenarios.

### Questions

* In the first sample, we implemented an Employee mixin for Person. Think about this implementation. If being an employee in a company is a context, what additional contexts could be there in the real world?
* In the first sample, the CoWorker mixin extends the Person class. Does it make sense to apply this mixin to a different class? If not, try to give a reason.
* After thinking about the answer to the question before. It seems that Person is “a fixed constant” and this class will be extended by mixins. Can you think of a scenario where the mixin is a bundle of functionality that can be added to more than just one class?

### Exercise 2 – Parameter

At the current state, we would have to set the properties of the base class Person (such as FirstName) after its creation via the object factory manually.

The next step is to add parameters to the at the object factory.

1. Add a new constructor with parameters to the person class.

public Person(string firstName, string lastName)

{

FirstName = firstName;

LastName = lastName;

}

1. Replace

var adama = ObjectFactory.Create<Person>(ParamList.Create());

with

var adama = ObjectFactory.Create<Person>(ParamList.Create("William", "Adama"));

### Questions

* In our sample, objects are created via a Factory Design Pattern. Can you explain the advantages of using Factories to instantiate classes over constructors within a few sentences? How would you try to convince a fellow developer to use factories instead of construction? Does this approach does also have disadvantages?
* If you use intellisense to look for possible parameters for ParamList.Create(), you will find that there are a lot of possibilities to choose. Can you explain the reason?

## Exercise 3 – SecurityContext

* 1. Add SecurityContext.cs

using System;

using MixinDomain.Interfaces;

using Remotion.Mixins;

namespace MixinDomain

{

[Extends(typeof(Person))]

public class SecurityRepresentativeMixin : Mixin<Person>, ISecurityRepresentative

{

public void GetWorkPackage()

{

Console.WriteLine(This.LastName + " has to inspect the flight deck for frakking cylon spies");

}

}

}

Interface

public interface ISecurityRepresentative

{

void GetWorkPackage();

}

public interface ISecurityRepresentative

{

void GetWorkPackage();

}

## Exercise 3 – A use case for „use“

Our first sample is a “extends” scenario. We have a fixed base class that can be extended with one or more mixins. There is also another scenario, where we have “a fixed mixin” that can be added to various classes. This scenario is called “use” scenario. In the following example we implement a mixin that will generate a string that summarizes all public properties of the class and their current values.

The idea is to generate a string with all public properties in a function (we will call it PropertyValues() of a mixin. This mixin is then used by various classes.

System.Console.WriteLine(((IPropertyValueProvider)employee).PropertyValues());

1. We add a new class to the domain and use the following code

using System.Reflection;

using System.Text;

using Remotion.Mixins;

namespace MixinDomain

{

public interface IPropertyValueProvider

{

string PropertyValues();

}

public class PropertyValueStringChainMixin<T> : Mixin<T>, IPropertyValueProvider where T : class

{

public string PropertyValues()

{

StringBuilder stringBuilder = new StringBuilder();

foreach (PropertyInfo propertyInfo in This.GetType().GetProperties())

{

stringBuilder.Append(propertyInfo.Name);

stringBuilder.Append(": ");

stringBuilder.Append(propertyInfo.GetValue(This, null));

stringBuilder.AppendLine();

}

return stringBuilder.ToString();

}

}

}

1. We also add the attribute to the class declaration of Person.

[Uses(typeof(PropertyValueStringChainMixin<Person>))]

### Questions

* We would not need to add a mixin to achieve this result. We could also add this method to the class Person too. What are the advantages to using mixins?
* Could you get the same advantages also with extension methods in this scenario?
* You might want to try out different parameters in the GetProperties method, such as

GetProperties(

BindingFlags.Public|BindingFlags.NonPublic|BindingFlags.Instance))

What happens?

* If you have tried it out, you will run into a StackOverflowExpection. Adding if (propertyInfo.Name != "PropertyValues") after the for clause does not solve the problem. Why?
* If you have solved the problem via if (!propertyInfo.Name.Contains("PropertyValues")) or something similar, you will see that the Salary is always 1000. Why?
* Can you explain what the following constructor does in detail? What is <T>?

public class PropertyValueStringChainMixin<T> : Mixin<T>, IPropertyValue where T : class

* Do you know what the keysword where means?
* Is it possible to change the constructor in the following way?

public class PropertyValueStringChainMixin : Mixin<object>, IPropertyValueProvider

* You might try to

System.Console.WriteLine(((IPropertyValueProvider)p).PropertyValues());

* System.Console.WriteLine(((IPropertyValueProvider) ((IEmployee)p)

).PropertyValues());

# Lab 2 Composition Pattern

## Exercise 1 – Adding CompositionPattern

### Task Add

* 1. Add a new Class Libarary
  2. Add File CompositionPattern.cs

using System;

using System.Linq;

using Remotion.Mixins;

using Remotion.Reflection;

namespace CompositionPatternSample

{

public interface IDomainObjectMixin

{

void OnTargetReferenceInitializing();

}

/// <summary>

/// Defines all framework requirements to a domain object.

/// </summary>

public interface IDomainObject

{

Guid ID { get; }

DomainObjectEventSource Events { get; }

}

public class DomainObjectEventSource

{

private readonly object \_sender;

public event EventHandler Committing;

public DomainObjectEventSource(object sender)

{

\_sender = sender;

}

public void OnCommitting()

{

var handler = Committing;

if (handler != null)

handler(\_sender, EventArgs.Empty);

}

}

/// <summary>

/// Acts as a convenience base class for domain objects in the mixin-based composition pattern. Implements the <see cref="IDomainObject"/> interface

/// and provides a <see cref="This"/> property that provides access to the composed interface.

/// </summary>

/// <typeparam name="TComposedInterface">The composed interface of the derived class.</typeparam>

public abstract class ComposedDomainObject<TComposedInterface> : IDomainObject

where TComposedInterface : class, IDomainObject

{

/// <summary>

/// Used to create instances of the composite domain object class. When the complete interface is automatically detected via

/// <see cref="ComposedDomainObject{TComposedInterface}"/>, it is automatically guaranteed (by the mixin engine) that the derived class has all

/// parts of the composed interface. Until then, the method checks that this is the case and throws an exception if otherwise.

/// </summary>

/// <typeparam name="TComposite">The type of the composite domain object.</typeparam>

/// <param name="ctorArgs">The constructor arguments.</param>

/// <returns>An instance of <see cref="TComposite"/>, cased to <see cref="TComposedInterface"/>.</returns>

protected static TComposedInterface NewObject<TComposite>(ParamList ctorArgs)

where TComposite : ComposedDomainObject<TComposedInterface>

{

var result = ObjectFactory.Create<TComposite>(ctorArgs) as TComposedInterface;

if (result == null)

{

var message = string.Format(

"Type '{0}' does not implement the complete interface '{1}'. Did you forget the CompleteInterfaceAttribute?",

typeof(TComposite),

typeof(TComposedInterface).Name);

throw new InvalidOperationException(message);

}

return result;

}

private readonly Guid \_id;

private readonly DomainObjectEventSource \_events;

protected ComposedDomainObject()

{

\_id = Guid.NewGuid();

\_events = new DomainObjectEventSource(this);

// ReSharper disable DoNotCallOverridableMethodsInConstructor

OnReferenceInitializing();

// ReSharper restore DoNotCallOverridableMethodsInConstructor

}

protected virtual void OnReferenceInitializing()

{

var mixinTarget = this as IMixinTarget;

if (mixinTarget != null)

{

foreach (var mixin in mixinTarget.Mixins.OfType<IDomainObjectMixin>())

mixin.OnTargetReferenceInitializing();

}

}

/// <summary>

/// Gets this <see cref="ComposedDomainObject{TComposedInterface}"/> instance, cast to the <see cref="TComposedInterface"/> type. This enables

/// callers to access the members of all composed mixins.

/// </summary>

/// <value>The this.</value>

protected TComposedInterface This

{

get

{

IDomainObject thisAsDomainObject = this;

return (TComposedInterface)thisAsDomainObject;

}

}

public Guid ID

{

get { return \_id; }

}

public DomainObjectEventSource Events

{

get { return \_events; }

}

}

}



### Task Discuss

## Task 2 – Add

Mixin.cs

using System;

using Remotion.Mixins;

namespace CompositionPatternSample

{

public interface IApprovable

{

bool Approved { get; set; }

string ApprovedBy { get; set; }

}

public interface IRevisionable

{

int RevisionNumber { get; set; }

}

public interface ITrackable

{

DateTime CreatedAt { get; set; }

}

public class TrackableMixin : Mixin<object>, ITrackable

{

public DateTime CreatedAt { get; set; }

public string CreatedBy { get; set; }

}

public class ApprovableMixin : Mixin<object>, IApprovable

{

public bool Approved { get; set; }

public string ApprovedBy { get; set; }

}

public class RevisionableMixin : Mixin<object>, IRevisionable

{

public int RevisionNumber { get; set; }

}

}

Add Impl:

using Remotion.Mixins;

using Remotion.Reflection;

namespace CompositionPatternSample

{

[CompleteInterface(typeof(Person))]

public interface IPerson : IDomainObject, ITrackable, IRevisionable, IPersonImpl

{

}

public interface IPersonImpl

{

string FirstName { get; set; }

string LastName { get; set; }

}

public interface IDocumentImpl

{

string DocumentName { get; set; }

}

[CompleteInterface(typeof(Document))]

public interface IDocument : IDomainObject, ITrackable, IApprovable, IRevisionable, IDocumentImpl

{

}

[Uses(typeof(TrackableMixin))]

[Uses(typeof(RevisionableMixin))]

public class Person : ComposedDomainObject<IPerson>, IPersonImpl

{

public static IPerson NewObject()

{

return NewObject<Person>(ParamList.Empty);

}

public string FirstName { get; set; }

public string LastName { get; set; }

}

[Uses(typeof(TrackableMixin))]

[Uses(typeof(ApprovableMixin))]

[Uses(typeof(RevisionableMixin))]

public class Document : ComposedDomainObject<IDocument>, IDocumentImpl

{

public static IDocument NewObject()

{

return NewObject<Document>(ParamList.Empty);

}

public string DocumentName { get; set; }

}

}

# Lab Summary

In this lab, you have successfully done the following

|  |
| --- |
| * You have created a mixin sample, where you have extended a class Person * You have used a mixin within the Person sample to collect property information * You have added parameter list to the construction. |

As you now are experienced in the main usage of mixins, you might want to dig deeper to become a real mixin expert. We suggest:

* Read “Head First Design Patterns” to be able to understand design techniques in total better. It helps you to put mixins in relation to other design topics
* Read the next re-motion HOL to understand another re-motion technology.
* On <http://en.wikipedia.org/wiki/Mixin> in further readings, there are many references to other mixin implementations, you might want to read them and compare them with the re-motion approach.
* Read the blogs on [www.re-motion.org](http://www.re-motion.org)
* For those who want to become real experts: Read “CLR via C#”, if you want to understand more in how code is generated and how mixins are generated. Take care: Reading the most essential parts book also with good background experience may take a lot of time. Plan this for an extended research time and prepare enough coffee.

# Common Practices

## Getting re-motion

For all examples, the re-motion binaries are required. You can either build them or download them via the build page (<https://www.re-motion.org/builds/>)

If you are not sure where to put the binaries, please read Adding references.

If you want to build re-motion, you have to get a version from the following subversion repository: <https://svn.re-motion.org/svn/Remotion/>.

In most cases, the trunk build is sufficient. If you do not want to work with a trunk build, you might want to get the build with that this HOL was tested: <https://svn.re-motion.org/svn/Remotion/tags/1.13.87>

Please read „How to build.txt“(found in the base directory of the repository) before your build.

## Adding references

There are several ways to add binaries in folder hierarchy in a project. We propose the following way for all re-motion HOLs:

Put them in a subfolder of your solution file alongside the subfolders for your projects:

* ./SolutionItems/References

Depending on the number of different assemblies and products you want to use on your project, you might want to add subdirectories for each product, too.

This has the advantage that you can also add other stuff such as Tools, Scripts, etc to your SolutionItems directory.