# Programmentwicklung Advanced II
Modulübergreifende Projektarbeit

## Inversion Of Control

**Ausgangslage**

Sie haben in Ihrer Applikation erfolgreich das Repository-Pattern implementiert.
Die Verwendung erfolgt so, dass das ViewModel eine Instanz des benötigten Repositories erzeugt und verwendet (oder so ähnlich – ggf. liegen noch weitere Layer zwischen ViewModel und Repository, dann verlagert sich die Verwendung des Repositories entsprechend in den dem Repository übergeordneten Layer – dies könnte z.B. ein BusinessLayer sein) – siehe Abbildung 1.

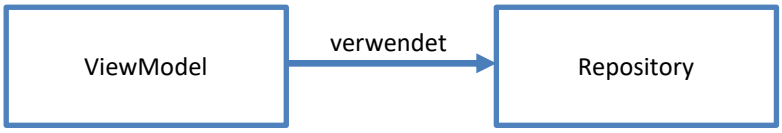


*Abbildung 1: Abhängigkeit zwischen ViewModel und Repository*

Es stellt sich die Frage: «Muss das ViewModel wirklich wissen, dass es das konkrete Repository verwendet?».
Im Normalfall besitzt das Repository auf irgendeine Art und Weise eine direkte oder indirekte (z.B. via ORM) Abhängigkeit zu einer Datenquelle. Das Mocking der Repository zwecks Testing der ViewModels (oder des entsprechenden Layers) erweist sich so jedoch eher schwierig. Das Problem entsteht durch die direkte Abhängigkeit zwischen ViewModel und Repository.
Wäre es nun nicht schöner, dass das ViewModel wüsste, dass es etwas verwendet, was das Verhalten des Repositories aufweist ohne zu wissen, welches Objekt schlussendlich effektiv das Verhalten implementiert? – Genau das kann mittels Inversion of Control erreicht werden.
Die entsprechende Struktur ist in Abbildung 2 ersichtlich.

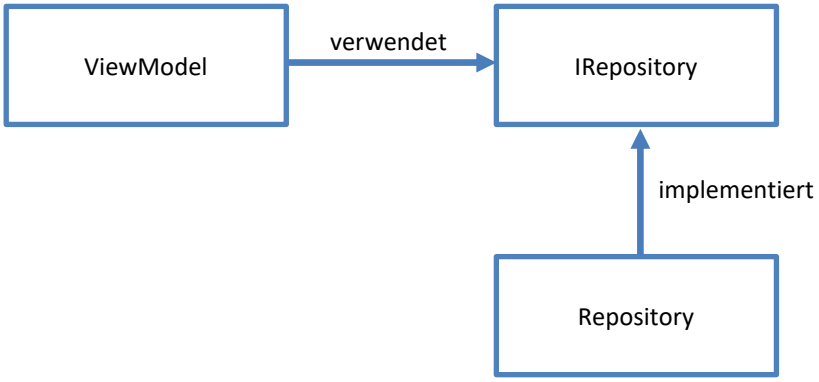


*Abbildung 2: Invertierte Abhängigkeit zwischen ViewModel und Repository*

Nun ist es so, dass die Klasse `Repository` das Interface `IRepository` implementiert. Die Klasse `ViewModel` verwendet eine Instanz von `IRepository` anstatt das konkrete Repository. So kann nun die konkrete Implementierung von `IRepository` beliebig ausgewechselt werden ohne die Klasse `ViewModel` anpassen zu müssen.

Nun stellt sich nur noch die Frage, wie wir nun zum Objekt gelangen, welches das Interface `IRepository` implementiert. Diese Aufgabe übernimmt nun das IoC-Container-Framework.

Dieses Framework ist in der Lage, uns das dem Interface entsprechende Objekt zur Verfügung zu stellen. Dies kann nun über zwei Wege erreicht werden:

1) Im `ViewModel` stellen wir dem IoC-Container-Framework die Anfrage, uns ein Objekt zur Verfügung zu stellen, welches das Interface `IRepository` implementiert
2) Wir lassen bereits das `ViewModel` vom IoC-Container-Framework erstellen. Dieses «injected» uns dann das konkrete Repository-Objekt z.B. im Constructor des `ViewModels`.

**Auftrag**

In Ihrer Applikation sollen die Repositories bzw. die ViewModels mittels Dependency Injection erstellt werden.

Listing 1 zeigt die grundlegende Struktur gemäss obiger Variante 2). Als IoC-Container-Framework wird in diesem Beispiel Autofac[1] verwendet. Zusätzlich wird Moq[2] für das Mocking der Objekte für UnitTesting verwendet. Es steht Ihnen frei, welche Frameworks Sie verwenden.

```
class Program{
  private static IContainer container;
  static void Main(string[] args) {
    /* ... */

    // Create the IoC-Framework-Container
    // In UnitTesting, use BuildTestingAutofacContainer()
    container = BuildAutofacContainer();
    //container = BuildTestingAutofacContainer();

    // Resolve LocationViewModel from Container
    // -> ILocationRepository will be automatically injected
    var vm = container.Resolve<LocationViewModel>();

    // Use the ViewModel
    Console.WriteLine(vm.GetLocationCount());
  }

  private static IContainer BuildAutofacContainer() {
    var builder = new ContainerBuilder();

    builder.RegisterType<LocationRepository>().As<ILocationRepository>();

    builder.RegisterAssemblyTypes(Assembly.GetExecutingAssembly())
        .Where(t => t.Name.EndsWith("ViewModel"));
    return builder.Build();
  }

  private static IContainer BuildTestingAutofacContainer() {
    var builder = new ContainerBuilder();

    // Build the Mock for ILocationRepository and register it
    var mock = new Mock<ILocationRepository>();
    mock.Setup(foo => foo.Count()).Returns(7);
    builder.RegisterInstance(mock.Object).As<ILocationRepository>();

    builder.RegisterAssemblyTypes(Assembly.GetExecutingAssembly())
        .Where(t => t.Name.EndsWith("ViewModel"));
    return builder.Build();
  }
}
```

*Listing 1: Grundstruktur Repository mit IoC-Framework (Fortsetzung siehe nächste Seite)*

[1] Autofac: https://www.nuget.org/packages/Autofac/
[2] Moq: https://www.nuget.org/packages/Moq/

```
public class LocationRepository : ILocationRepository {
  public long Count() {
    /* ... */
    return ctx.GetTable<Location>().Count();
  }
  /* ... */
}

public interface ILocationRepository{
  long Count();
  /* ... */
}

public class LocationViewModel {
  private readonly ILocationRepository locationRepository;

  // Inject the Repository with Constructor Injection
  public LocationViewModel(ILocationRepository locationRepository) {
    this.locationRepository = locationRepository;
  }

  /* ... */
  public long GetLocationCount() {
    return this.locationRepository.Count();
  }
}
```

*Listing 1: Grundstruktur Repository mit IoC-Framework (Fortsetzung)*

Erstellen Sie in Ihrer Solution zusätzlich ein Testprojekt, welches für alle Repositories entsprechende Mocks erstellt.

**Lieferergebnis**

Als Abgabe wird der komplette Sourcecode als ZIP-Datei oder Link zu Ihrem GitHub-Repository erwartet. Sofern Sie den Code als GitHub-Repository abgeben denken Sie daran, einen entsprechenden Tag zu erstellen und diesen in Ihrer Abgabe zu vermerken.

Gemäss Auftrag soll Ihre Solution ein Testprojekt enthalten. Dieses Testet die komplette Business-Logik Ihrer Applikation. Für die verwendete Repositories werden Mocks erstellt und entsprechend verwendet. Dieses wird für die Bewertung verwendet. Vermerken Sie die nötige Handhabung in Ihrer Kurzdokumentation.

Denken Sie weiter daran, dass Ihr Projekt alle Abhängigkeiten enthält bzw. diese nach einem GitHub-Checkout beim ersten Build wiederherstellt.

**Bewertung**

Gemäss Initialdokument

**Termin**

Gemäss Moodle-Abgabe