Proces izrade biblioteke za mapiranje objekata u C#-u

Romić, Ante

JOSIP JURAJ STROSSMAYER UNIVERSITY OF OSIJEK

Department of Mathematics

University Undergraduate Study in Mathematics and Computer Science

Development process of making an object mapping library in C#

Bachelor's thesis

Mentor:

Domagoj Matijević, Ph.D.

Co-Mentor:

Nathan Chappell

Candidate:

Ante Romić

Osijek, 2023

Acknowledgments

I would like to thank my mentor and my supervisor (co-mentor) from my workplace for advising me during the writing of this thesis. Special thanks to my co-mentor, considering that he didn't have to do any of it, I think that he had a great impact on this work. He was very resourceful, and offered me great guidance and feedback during the progress of this project and the thesis. The practical part of this project was very interesting to work on, the thesis part a bit less, but we managed to work it out and I feel relieved that this is finally done.

The author used ChatGPT to aide in writing this paper.

- The generative technology was used to perform corrections, improve tone, and brainstorm ideas.
- No generated information factual in nature or related to the topic of the publication has been represented as the author's original work.
- The author takes ultimate responsibility for the content of this publication.

Contents

1 Introduction **5**

2 C# **7**
2.1 Intermediate Language 8
2.1.1 Case Study: Building a Type at Runtime 8
2.2 Code Generation 10
2.3 Reflection 11
2.4 Related Work 14

3 Mapper **17**
3.1 Problem Statement 17
3.2 The set $\mathcal{R}$ 18
3.2.1 Mappings in $\mathcal{R}$ 19
3.2.2 Mappings in $\mathcal{C}\#$ 20

4 Testing **21**
4.1 Testing in this project 21
4.2 Functionality Tests 21
4.2.1 Program Correctness 21
4.2.2 Aiding Development 22
4.3 Performance Tests 22

5 Conclusion **25**

Literature **27**

CV **29**

1 | Introduction

In this thesis we will give some insight into the development process of making an object mapping library in C#. After we talk about C# in general, we are going to describe the concepts and the tools that would make the core of our project. These would be Intermediate Language (IL), code generation, reflection and for our testing part of the development: NUnit for integration tests and unit tests, BenchmarkDotNet for our performance tests. The goal of this project was to make an object mapping library that would create a new object of the target type with the property values derived from the source type. The main use case for this kind of library would be within a web development project, how and why will be described in the thesis. We will also define mapping relationships, meaning that we will explain what types can be mapped from and mapped to within the library.

The inspiration to develop this kind of library came to me after working with a similar tool called AutoMapper. Some of its functionalities were interesting to me, so I wanted to find out more about them. I especially wanted to learn more about reflection which is the core concept of AutoMapper. Also, there were a few cases where AutoMapper would not behave as I would expect it to, so I wanted to make my own mapper that would work in a different way.

2 | C#

C# (pronounced "See Sharp" [4]) is a multi-paradigm programming language used in large-scale enterprise application development. It is called a component-oriented language, referring to the fact that C# is a part of a larger system, known as the Common Language Infrastructure (CLI), and C# is typically compiled into a file called an *assembly*. An assembly is a binary file (a specialization of the Portable Executable file format) that describes types, resources, and can contain executable intermediate language (IL) code.

This IL is an assembler-like language built to expose the Virtual Execution System (VES) to compiler writers. The syntax[1] and semantics of the IL is given in [9]. Conceptually the most important thing to understand is that C# is a high level language that compiles down to IL. IL is a portable-binary format which describes *metadata*. This metadata can be interpreted by the VES. The VES is essentially a computer, but encoded in software. This computer (VES) has state and interprets instructions (IL) which can alter this state. IL is typically not viewed in its binary format, but as an assembler like language called *ILasm*. We will sometimes refer to ILasm as IL, blurring the distinction (there *should* be a surjective mapping from valid ILasm programs to valid IL).

Throughout this paper we will not typically distinguish between C# and the CLI. In particular, we will not fuss about whether some feature that is present in the system as it is practically used is prescribed by the language or some other specification. While it is certainly not irrelevant in general, it is not relevant to our current concerns, other than to understand the system better.

One of the most powerful features of C# is known as *reflection*, or the ability to introspect the program at runtime. Reflection offer the capability to, for example:

1. Inspect what constructors are available for a type
2. Inspect what the parameter types, names, and positions are for such a constructor
3. Invoke the constructor with arguments determined at runtime
4. Inspect what properties are defined on a type

Another very powerful feature available to the runtime through reflection is the ability to dynamically create types, methods, even entire assemblies. This enables a very convenient interface for *code generation*. While an advanced feature of

[1] In reality, the syntax of the IL assembler language is given by the current implementation.

the language, code generation offers opportunities to manually implement optimizations that would be difficult or impossible for the compiler to do, in particular optimizations which cannot be done before runtime. The use case for this project, and similar mapping technologies, is to use IL generation to create very efficient methods based off of information captured by the traditional reflection capabilities. The results discussed in 4.3 demonstrate these performance discrepancies.

2.1 Intermediate Language

Intermediate Language is a product of compilation of the code written in high-level .NET languages. When the code that is written in one of these languages is compiled, the source code is transformed into an object file that contains enough information for your operating system's runtime-loader to execute your program. These object files contain machine-code - a sequence of bytes which represent instructions for the processor to execute.

Using IL allows you to fetch and store values from and to memory, call methods available in properly referenced assemblies and gives you the ability to manipulate a stack. One purpose of having such a system is portability: theoretically, an assembly compiled on any machine should run on any other machine for which there is a conforming implementation of the CLI (e.g. your dotnet program built on windows will run with dotnet on linux)[8].

2.1.1 Case Study: Building a Type at Runtime

During the implementation of the mapping tool, I implementing the logic for mapping class type properties where I would create a new instance of the type if it was originally null on the destination object that we were mapping the data to. Then came the question of what to do if the property type is an interface and there were 2 options.

Since an instance of an interface can't be created, the first option was to simply ignore the property if its type was an interface and say that the mapper does not support interface types. On the other hand, after using Automapper(2.4), I knew that they have interfaces handled somehow, so that had to be possible in C#. After doing some research on how to do it, I came across a class called TypeBuilder.

The TypeBuilder class contained in C# `System.Reflection.Emit` namespace can be used to define and create new instances of classes during run time. Microsoft's documentation had a few good examples of how to use it.
`System.Reflection.Emit` also contains classes like `MethodBuilder`, `FieldBuilder` and `PropertyBuilder` which were all used in order to create a new type. In order to use the TypeBuilder class, a new dynamic assembly must be created along with a dynamic module for it. Consider the `interfaceType` the interface from which we want to create our new type.

```
1     var assemblyName = new AssemblyName($"{interfaceType.Name}
    Assembly");
```

```
2     var assemblyBuilder = AssemblyBuilder.DefineDynamicAssembly(
    assemblyName, AssemblyBuilderAccess.Run);
3     var moduleBuilder = assemblyBuilder.DefineDynamicModule("
    DynamicModule");
4     var typeName = $"{interfaceType.Name}_{Guid.NewGuid():N}";
5     var typeBuilder = moduleBuilder.DefineType(typeName,
    TypeAttributes.Public);
6
7     typeBuilder.AddInterfaceImplementation(interfaceType);
```

After collecting all the properties of the specified interface, for each of the properties we define a property and a field on our new type.

```
1     var fieldName = $"<{property.Name}>proxy";
2     var propertyBuilder = typeBuilder.DefineProperty(property.Name,
      PropertyAttributes.None, property.PropertyType, Type.EmptyTypes
    );
3     var fieldBuilder = typeBuilder.DefineField(fieldName, property.
    PropertyType, FieldAttributes.Private);
```

After doing that, what's left is to define a get and set method for each of the properties. In the end, after calling the `CreateType()` on our previously defined `typeBuilder` we got our newly created type with all the properties the specified interface has.

```
1 //using Microsoft Intermediate Language Generator define the get
    method of the property.
2 private static MethodBuilder BuildGetter(TypeBuilder typeBuilder,
    PropertyInfo property, FieldBuilder fieldBuilder,
    MethodAttributes attributes)
3 {
4     var getterBuilder = typeBuilder.DefineMethod($"get_{property.
    Name}", attributes, property.PropertyType, Type.EmptyTypes);
5     var ilGenerator = getterBuilder.GetILGenerator();
6
7     ilGenerator.Emit(OpCodes.Ldarg_0);
8     ilGenerator.Emit(OpCodes.Ldfld, fieldBuilder);
9     ilGenerator.Emit(OpCodes.Ret);
10
11    return getterBuilder;
12 }
13
14 //using Microsoft Intermediate Language Generator define the set
    method of the property.
15 private static MethodBuilder BuildSetter(TypeBuilder typeBuilder,
    PropertyInfo property, FieldBuilder fieldBuilder,
    MethodAttributes attributes)
16 {
17    var setterBuilder = typeBuilder.DefineMethod($"set_{property.
    Name}", attributes, null, new Type[] { property.PropertyType });
18    var ilGenerator = setterBuilder.GetILGenerator();
19
```

```
20      ilGenerator.Emit(OpCodes.Ldarg_0);
21      ilGenerator.Emit(OpCodes.Ldarg_1);
22      ilGenerator.Emit(OpCodes.Stfld, fieldBuilder);
23      ilGenerator.Emit(OpCodes.Ret);
24
25      return setterBuilder;
26  }
```

Defining the set and get method for these properties was done with the help of Microsoft's Intermediate Language Generator (ILGenerator) which was used to define the bodies for our get and set methods.

2.2 Code Generation

While code generation is not the main focus of this project, it came up in two places and is worth a mention. It was used during the dynamic creation of types at runtime (see 2.1.1), and it also occurs when creating methods for the `ILMapper`. Wikipedia offers the following definition:

> In computing, code generation denotes software techniques or systems that generate program code which may then be used independently of the generator system in a runtime environment. [12]

Succinctly, *code generation* occurs whenever a program writes a program. The author likes to consider the following as paradigms for code-generation:

1. *Ad-hoc*: when a code generation occurs in a way that is not reusable or modular
2. *Template-based*: when code generation occurs based on static templates
3. *Structured*: when code generation occurs in a composable manner

The main axes of variation for these different techniques are (unsurprisingly) development-effort and reusability.

Ad-hoc code generation typically involves one-shot scripts, or, for example, a command executed in a REPL[2]. A concrete example is the use of a web-browser to inspect an HTML `<table>` element, and using JavaScript to iterate through the rows and output XML elements in the browser's console. Such techniques are typically not repeatable, which means that if the same task is required in the future, the whole process must be repeated.

Template-based code generation uses some sort of string-replacement engine to aid in generation. This approach is typically suitable when the overall structure of the content to be generated is the same for every target, and the process can be

[2] read-eval-print loop

parameterized by some simple data structure. Two well-known template-engines include Jinja for Python and TextTransform for C#. These engines offer the ability to execute code specified in the templates themselves, meaning that the templates actually specify a *mini-language*. Using template engines adds complexity in one area, but can make certain tasks very convenient.

Structured code generation typically will occur when you need your code generation schemes to be composable. Often times the generation logic will correspond to the abstract syntax of the source or target of generation. For example, the generation of IL for C# is handled in a very structured manner, to the point that there is an object-oriented interface to emit every single opcode available. This is required in part simply due to the complexity of the target (an assembly).

In this project we take an *ad-hoc* approach. The IL generation occurs in controlled settings for a specific purpose, and composability and reuse has not been a design consideration. For the type-building, the amount of generation required is straightforward and a small amount of special purpose code needs to be written. However, it could be interesting to consider optimizations (e.g. inlining) for the ILMapper. To effectively implement such optimizations would require reconsidering the current generation process.

2.3 Reflection

Reflection is the mechanism of discovering an assembly content at runtime including an assembly's metadata, its classes, class members, their types and scope. It can be used to retrieve data that can not be accessed before runtime, it provides various classes like MethodInfo, PropertyInfo, FieldInfo, ConstructorInfo and EventInfo that allow you to access details about methods, properties, fields, constructors and events of a type.

```
1 void GetTypeInfo(Type type)
2 {
3     PropertyInfo[] propertyInfo = type.GetProperties();
4     ConstructorInfo[] constructorInfo = type.GetConstructors();
5     EventInfo[] eventInfo = type.GetEvents();
6     MethodInfo[] methodInfo = type.GetMethods();
7     FieldInfo[] fieldInfo = type.GetFields();
8 }
```

Listing 2.1: Retrieving type info using reflection

It can also be described as a process of collecting information on the program's features and operating on itself. It involves the ability of a program to "reflect" upon itself and that would be where the term "Reflection" comes from. This feature is particularly useful if we need to extract the program's metadata and modify its behavior.

A lot of information can be found on these objects collected above, we can inspect if the property has a get or set method defined, check its accessibility level,

get or set its value. We can use reflection to create instances of objects, invoke methods, even set the generic types of the method dynamically, instantiate types dynamically. Reflection is commonly used in scenarios like dynamic loading of assemblies, creating plugins, or building tools like object mappers, serializers, and dependency injection containers. In our case we used it to build an object mapper. Reflection is primarily handled trough the C# `System.Reflection` namespace which contains classes and methods used to retrieve metadata and runtime type information.

In C# every type is derived from `System.Object`, so we can use object to reference instances of any type and use it as the base type when working with reflection, considering that we won't know the actual type of the object until the runtime. Lets take a look at the following class.

```
1  public class ExampleClass
2  {
3      public ExampleClass(string firstName, string lastName)
4      {
5          FirstName = firstName;
6          LastName = lastName;
7      }
8      public ExampleClass() { }
9
10     public string FirstName { get; set; }
11     public string LastName { get; set; }
12 }
```

Listing 2.2: Class example

If we have an object instance of that class and want to create a "Deep Copy" of that object, we want to create a new object of that class with the same property values as our original object, but with a different reference. We can do that using reflection and here are 2 examples how to do it.

```
1 void CreateDeepCopy(object exampleObject)
2 {
3     var type = exampleObject.GetType();
4     var properties = type.GetProperties();
5
6     //Approach 1
7     var instance1 = Activator.CreateInstance(type);
8     foreach(var property in properties)
9     {
10        property.SetValue(instance1,
11            property.GetValue(exampleObject));
12    }
13
14    //Approach 2
15    var propertyValues = properties.Select(prop =>
16        prop.GetValue(exampleObject)).ToArray();
17    var constructors = type.GetConstructors();
18    var instance2 = constructors.First().Invoke(propertyValues);
19 }
```

Listing 2.3: Creating a deep copy of an object

In the first approach we created an instance of the type by using the `Activator.CreateInstance` method. `Activator` class contained in the `System` namespace enables you to create instances of the specified types dynamically and this is particularly useful when we don't know the type until runtime. It uses the default constructor to create it. This approach would not work for class types without an empty constructor. Afterwards we iterated trough the types properties, and set the property value on `instance1` to the value from `exampleObject` for the same property.

In the second approach, we first collected all the property values from our `exampleObject`. Calling `type.GetConstructors()` returns the type's constructors in the declaring order, meaning that the non-empty constructor will be the first element in the array. After that we just needed to invoke that constructor with the property values passed to it and we got a new instance of that type. If we had a class `ClassA` that had class object properties instead of the simple strings that we have in our `ExampleClass`, in order to create a deep copy of the `ClassA` object, we would need to create a "deep copy" for each property.

While implementing the mapper, I had a case where my type was an interface, but I needed an instance of that type. Since an instance of an interface cannot be created the solution to this problem was to first create a new type that contains all the properties that this interface has, along with the properties it may inherit and then create an instance of that type. I've managed to do this using reflection along with ILGenerator.

Type.GetInterfaces() is another useful method in `System.Reflection` namespace that helped us with this cause. It returns all the interfaces that the type inherits or implements, but in this case we just needed to know the inherited ones since our type was already an interface.

While reflection is very useful in some cases, the one using it needs to be very careful since it can have performance overhead and the code could be very hard to maintain if it is not correctly used. By using reflection, one of the things that C# is known to be especially good at is lost and that is the compiler's type safety checking. This would be one of the good reasons not to use reflection if you are not a 100% sure that you need it.

Reflection is also an available feature in other programming languages. Python, for example, has a concept called *introspection* that defines the ability to find out information about object at runtime. Reflection in Python then enables those objects to be modified, so we could say that Python's *introspection* and *reflection* together would make something similar to what reflection is in C#. On the other hand, Javascript contains the `Reflect` namespace object containing static methods for invoking interceptable JavaScript object internal methods. Almost every `Reflect` method has a corresponding method that can be used with some other syntax. Reflection in Java is somewhat similar to reflection in C#, `Java.Lang.Reflect` contains methods similar to C#'s `System.Reflection` methods. All these languages have the *reflection* feature, all used for the similar purpose, just with a different syntax and some functionalities. Pascal and C are some of the programming languages that do not provide this feature.

2.4 Related Work

Our use of inductive (i.e. recursive) definitions for defining the *maps-to* relation is guided by [10], and we highly recommend it as an introductory text for lambda calculus and type systems. The full formalization of our relation would be to take the minimal fixed point as described in [10], such matters are left as future work (see **??**).

There are other similar tools to this one and one of them is the most interesting to us because it inspired me to do this work. That tool is called AutoMapper. Automapper is a library mostly used in web development with the purpose of mapping objects from one type to another. What makes AutoMapper interesting is that it provides some interesting conventions to take the dirty work out of figuring out how to map type A to type B [6].

It has a feature of setting custom configurations between mapping two different types, but as long as type B follows AutoMapper's established convention, almost zero configuration is needed to map two types. It simplifies the often repetitive and error-prone task of mapping data between different object types and streamlines the process of data transformation, allowing the user to focus more on the application's core logic and less on manual object mapping.

Mapping can occur in many places in an application, but mostly in the boundaries between layers, such as between the UI/Domain layers, or Service/Domain layers. Concerns of one layer often conflict with concerns in another, so object to object mapping leads to segregated models, where concerns for each layer can affect only types in that layer.

AutoMapper uses reflection in order to make most of its functionalities work. We can say that its base concept is reflection. It also uses a lot of `expressions`

in its implementation. An expression is a sequence of operators and operands. This clause defines the syntax, order of evaluation of operands and operators, and meaning of expressions [2].

Other tools similar to AutoMapper would be Mapster, Mapperly, Mapping Generator and ExpressMapper. All of these tools mostly serve for the same cause - object to object mapping, with some differences in their implementation, their performances and the features that they provide.

3 | Mapper

3.1 Problem Statement

This library is designed to be used for mapping an object of one type into another type in C#. This tool is used to create a new object of the set destination type with the property values from source being mapped to the newly created object.

It includes a map function

```
public object? Map(object? from, Type fromType, Type toType)
```

that returns a newly created object of type `toType` by getting the values of the properties from object `from` of type `fromType` and using them to create the returned object.

The library contains 2 different "mappers", both implementing the Map method previously mentioned. One is called `ReflectionOnlyMapper` implemented mostly using reflection to do the mapping and the other one called `ILMapper`.

How `ILMapper` works in simple terms is that it first gets the properties of the source object and tries to find a constructor of the `toType` that can be called after altering property values using the IL and code generation. The mapping will then succeed if the `fromType` is mappable to `toType`(see **??**).

The reflection only mapper first creates a new instance of the target type. How that is done depends if the type is a record type, an interface or a class type. Then by iterating trough the destination type properties, recursively maps the values from the source properties with the same name to the newly created object.

If we have the 2 following test classes:

```
public class TestClassFrom
{
    public TestClassFrom() { }
    public string Name { get; set; }
    public int Number { get; set; }
}
public class TestClassTo
{
    public TestClassTo() { }
    public string Name { get; set; }
    public int Number { get; set; }
```

```
}
```

using one of the mappers would look something like this.

```
public void MapperUseCase()
{
    var mapper = new ILMapper(); //or var mapper = new
    ReflectionOnlyMapper();
    TestClassFrom from = new TestClassFrom()
        { Name = "test" , Number = 1 };
    TestClassTo to = (TestClassTo)mapper
        .Map(from, typeof(TestClassFrom), typeof(TestClassTo));
}
```

In the end, on our `to` object we would have the property "Name" with the value "test" and the property "Number" with the value 1.

A tool like this would be especially useful in web development or any other multi-layered architecture project where we have different type models used in different layers of the application. Using this tool would here help us "move" the data between the layers. We would first map the data to a type used in another layer whose method will then be called. This is done to prevent unnecessary information being passed between these layers.

The implementation of this library can be seen using the following link.

3.2 The set $\mathcal{R}$

The rest of this chapter is dedicated to defining the *maps-to* relationship formally.

A very obvious problem when designing a generic mapping library is describing what types can be *mapped-to*. We decided to inductively define a set of types we call $\mathcal{R}$ (the $\mathcal{R}$ is for `record`, a keyword of C#). The basic idea is to allow some *primitive* or *built-in* types, then describe ways to construct new types from them. This is precisely what the C# specification already does, but we will describe a much smaller set of types than that permitted by C#. We will denote the full set of C# types as $\mathcal{C}\#$. We denote Id the set of valid C# *identifiers*.

The inductive definition starts off with the rules for the built-in types:

$$\frac{}{String : \mathcal{R}} \qquad \frac{}{[U]Int\{8, 16, 32, 64\} : \mathcal{R}}$$

$$\frac{}{Double : \mathcal{R}} \qquad \frac{}{Bool : \mathcal{R}}$$

These axioms are the basis for our type system. We use some shorthand notation to refer to the signed and unsigned integral types, as there are a lot of them. We will use the following in the sequel:

$$\mathcal{Z} = [U]Int\{8, 16, 32, 64\}$$

$\mathcal{Z}$ should be reminiscent of the set of integers $\mathbb{Z}$. Informally, it can be thought of as the disjoint-union of all the types denoted, and when we write $T : \mathcal{Z}$ we mean that T is one of those types. Note that *Bool* is not included.

The next set of rules describes how we build more complex types from the built-in ones:

$$\frac{T : \mathcal{R}}{T[] : \mathcal{R}} \text{ array} \qquad \frac{T_{1..n} : \mathcal{R},\ N_{0..n} : Id, \quad 1 \leq i < j \Rightarrow N_i \neq N_j}{N_0(T_1N_1, T_2N_2, \ldots, T_nN_n) : \mathcal{R}} \text{ record}$$

These rules state that we can take an existing type from $\mathcal{R}$, and create an array type out of it, or use it in a `record` definition. The main awkwardness in the record clause is ensuring that a name is not repeated in the type definition. There is likely a more elegant way to do it, but it's not the most important part either.

With $\mathcal{R}$ formally defined, we can move on to the *mappable-to* relation.

3.2.1 Mappings in $\mathcal{R}$

Given two types, $S, D : \mathcal{R}$ (Source, Destination), we want to know if we can *map S to D*. This calls for defining a relation on $\mathcal{R}$. It seems intuitively that we are looking for a transitive, reflexive order on $\mathcal{R}$, since we'd essentially like to take a thin subcategory of $\mathcal{C}\#$, where a mapping from one type to another creates a morphism in $\mathcal{R}$. But we first must define the order, then we actually need to prove it satisfies the required properties.

We will use the symbol $\mapsto$ to indicate *maps to*, that is $S \mapsto D$ is read: *S maps to D*. We first state how primitive types map to one another.

$$\frac{T : \mathcal{R}}{T \mapsto String} \qquad \frac{T : \mathcal{Z}, U : \mathcal{Z}}{T \mapsto U \quad U \mapsto T}$$

$$\frac{}{Bool \mapsto Bool} \qquad \frac{T : \mathcal{Z}}{T \mapsto Double}$$

This behavior is mainly taken from the C# runtimes behavior with respect to implicit casting. Also, since everything in C# has a `ToString()` method, it seems reasonable to use it to map anything to a *String*.

The rule for arrays is straightforward:

$$\frac{T \mapsto U}{T[] \mapsto U[]}$$

Intuitively, we iteratively map each element.

Finally, for records, we just need to make sure that for every name that appears in the *destination type* (D), we have a corresponding name in the *source type* (S), and that the type that corresponds to the name in S *maps to* the type corresponding to the name in D. Clearly, we need a syntax to express all this.

Without making a big fuss, for $T : \mathcal{R}$ we'll write $N \in T$ to mean that $N : Id$, and N is the name of a property of T. Furthermore, we write $T.N$ to denote the type of that property. For example, let $T = Pair(Int32\ Item1, String\ Item2)$. Then $Item1 \in T$ and $Item2 \in T$, and $T.Item1 = Int32$, $T.Item2 = String$.

With this notation, we can state our rules for *records*:

$$\frac{S, D : \mathcal{R},\ (\forall N \in D)\, S.N \mapsto D.N}{S \mapsto D}$$

It's worth noting that the rules for *arrays* and *records* are very similar for what might be found in an inductive definition of the rules for *subtyping* or *assignability*, but all three of these relationships are distinct from one another.

3.2.2 Mappings in $\mathcal{C}\#$

In general, to map from a type $S : \mathcal{C}\#$ to a type $D : \mathcal{R}$, we look at the properties of S with public getters. As before, for $S : \mathcal{C}\#$ we write $N \in S$ to mean that N is the name of a public property of S and $S.N$ for its type. Noting that there is an obvious injection $\mathcal{R} \to \mathcal{C}\#$, we extend our type rules with:

$$\frac{S : \mathcal{C}\#,\ D : \mathcal{R},\ (\forall N \in D)\, S.N \mapsto D.N}{S \mapsto D}$$

4 | Testing

4.1 Testing in this project

There are three distinct cases for testing in this project: aiding development, partially proving correctness, and demonstrating effective performance. There are also two distinct types of tests that we run: *functionality tests*, and *performance tests*. This table shows how these are all related:

	development	correctness	performance
functionality tests	✓	✓	
performance tests	✓		✓

4.2 Functionality Tests

4.2.1 Program Correctness

The case of using tests to demonstrate partial correctness is quite obvious (you have pretty convincing evidence that your program works on the test cases at least!), but it should be mentioned that a simple mathematical specification of a program makes it possible to somewhat formally prove correctness by relating lines of code to steps of an algorithm. To take the process further would be to decompose the program into relevant modules and create an abstract interpretation of interacting subsystems. Then correctness can be demonstrated in the interpretation, and depending on how much effort is put into the analysis of the programming artifacts, some amount of confidence in correctness can be assigned to the original program.

Due to expressed interest, there is ongoing development of the project, so such a formal approach to correctness is left as future work.

4.2.2 Aiding Development

We used test-driven development (TDD) throughout the development process.

> Test-driven development (TDD) is a software development process relying on software requirements being converted to test cases before software is fully developed. [13]

The typical development process is as follows[11]:

1. Design the interface for the functionality you need
2. Write a test case using the interface
3. Implement the interface until the test passes

One advantage of TDD is that you tend to make better interfaces, since you want the smallest possible surface for testing. Another is that you are able to refactor your code and make changes with some confidence that things are still working. Having tests while developing can also help catch bugs and design flaws early on.

4.3 Performance Tests

This project was redeveloped with the intention of being able to compare the performance of reflection-methods with IL generation. In particular, a comparison of our "simple" IL generation, the straightforward application of reflection, a professional tool (`AutoMapper`), and a trivial "hand-mapping" were to be compared. Here is the "hand-mapping" we used:

```
1 var mapper = (PairPointFrom from) =>
2       new PairPointTo(
3         new((short)from.L.X, (short)from.L.Y),
4         new((short)from.R.X, (short)from.R.Y)
5       );
```

As you can see from the code, this "mapper" is just an anonymous function that directly constructs the mapped-to object with appropriate properties and conversions annotated directly. The reason to include a test for this case is to see if the mapping tools can outperform what a user can create by hand.

In general, performance testing is considered to be a very tricky task. We used a library called BenchmarkDotNet[7]. We ran into the issue of coming up with some common types that could be mapped from and to with all the technologies of interest. We finally used a rather simple data-structure, with one level of nesting. We decided this would be suitable as empirical evidence to support our intuition about relative performance of the different mappers. The final results we ended up with are recorded in table 4.1.

Method	Mean	Error	StdDev
`LambdaMapper`	274.0 us	1.61 us	1.43 us
`AutoMapper`	1,209.6 us	5.41 us	5.06 us
`ILMapper`	5,621.9 us	16.38 us	15.32 us
`DummyMapper`	5,661.7 us	22.74 us	21.27 us
`ReflectionOnlyMapper`	17,125.1 us	87.41 us	81.77 us

Table 4.1: Performance testing results

The `LambdaMapper` refers to the mapper created with an anonymous function, `AutoMapper` refers to using that library, the `DummyMapper` is the same as the `ILMapper`, but with no use of reflection (it was used to test the IL generation before the reflection component was implemented), and the `ReflectionOnlyMapper` is a mapper that runs purely on reflection.

The results seem to confirm the hypothesis that the use of IL generation gives a significant performance advantage over using reflection, however our implementation has a lot of room to improve (compare to AutoMapper). We should also observe how fast the "hand-written" `LambdaMapper` is - it could be interesting to investigate precisely why.

5 | Conclusion

This thesis provides insights into the development process of a mapping library in C#. It includes both examples and description of implementation and the theoretical part behind it. The core functionality of the library revolves around creating a new object of the target type, with its property values derived from an object of the source type.

From these discussions, we can draw several important conclusions:

1. **Practical Application**: The mapping library serves as a practical solution for transforming objects from one type to another. It is particularly useful in scenarios where different data models are employed across various layers of multi-layered applications, such as web development projects.

2. **Choice of Mapping Strategies**: The library offers two distinct mapping strategies: the 'ReflectionOnlyMapper' and the 'ILMapper.' This choice of strategies allows developers to select the most appropriate approach based on their specific mapping requirements and performance considerations.

3. **Use of Reflection and Code Generation**: The library employs reflection and code generation techniques to achieve its mapping objectives. Reflection is used by the 'ILMapper' to get information about constructors and properties while the 'ReflectionOnlyMapper' leverages code generation construct new types.

4. **Formalized Mapping Relationships**: In the thesis, we discuss the creation of a structured system, referred to as $\mathcal{R}$ which serves to define how different types can be mapped to each other within the library. This system establishes clear guidelines for how these mappings should work, aiming to build a consistent and logical structure for understanding how the library operates. In simpler terms, it's like setting up a set of rules to make sure that different types can be transformed into one another in a systematic and predictable way, giving the library a solid mathematical foundation for its functionality.

5. **Ongoing Development**: The library is a work in progress, with the potential for further research and development to expand its capabilities. Also, there are possibilities for some code and performance optimizations.

Literature

[1] Assemblies in .NET learn.microsoft.com

[2] C# Expressions learn.microsoft.com

[3] TypeBuilder learn.microsoft.com

[4] A tour of the C# language learn.microsoft.com

[5] Dependency inversion learn.microsoft.com

[6] AutoMapper documentation automapper.org

[7] BenchmarkDotNet github repository github

[8] Intro to the CLI mono.software

[9] Standard ECMA-335 (6^{th} edition) ECMA International

[10] Types and Programming Languages Pierce, Benjamin C, *MIT Press*

[11] Working Effectively with Legacy Code Feathers, Michael C. *Prentice Hall*

[12] Code generation Wikipedia

[13] Test-driven development Wikipedia

CV

I was born in the city of Vinkovci, Croatia in 1999. When I was 3 years old I moved to the city of Ilok where I went to primary school and high school. After I graduated from high school in 2017. I enrolled in the Mathematics and Computer Science undergraduate study at the Mathematics Department of the Josip Juraj Strossmayer University in Osijek.