

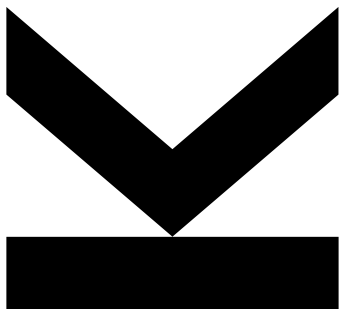
Author
Manuel Aumeier

Submission
Institute for
System Software

Thesis Supervisor
Dipl.-Ing Dr. Markus
Weninger, BSc

April 2026

Instrumenting C# LINQ Queries to Collect Execution Traces for Visualization



Bachelor's Thesis
to confer the academic degree of
Bachelor of Science
in the Bachelor's Program
Informatik

Bachelor's Thesis
**Instrumenting C# LINQ Queries to Collect Execution Traces
For Visualization**

Dipl.-Ing. Dr. Markus Weninger, BSc
Institute for System Software
T +43-732-2468-4361
markus.weninger@jku.at

Student: Manuel Aumeier
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: November 2025

Background and Motivation

C#'s Language Integrated Query (LINQ) is a powerful feature that enables developers to process data sequences in a declarative, functional style. It provides a high-level abstraction for complex operations on data collections, such as filtering, mapping, and aggregating. Despite its advantages, the underlying complexity of these pipeline operations can make it challenging for developers and students to fully comprehend how data flows through the system.

To address similar challenges in Java, prior work has successfully developed an automated marble diagram visualizer for the Java Stream API. This tool captures and analyzes the execution traces of stream operations. By tracking the complete journey of every input object through the stream pipeline, the system generates dynamic marble diagrams that visually represent data flow, item transformations, and operation sequencing.

Objective

The primary objective of this Bachelor's thesis is to extend this visualization capability to C# by instrumenting LINQ queries to generate execution traces compatible with the existing visualization system. While the underlying trace format may require slight modifications to account for the architectural differences between Java Streams and C# LINQ, cross-compatibility must be maintained to ensure the visualizer continues to work seamlessly for both languages.

Specific Tasks

To achieve these goals, the student will:

- Investigate and document the conceptual differences between Java Streams and C# LINQ.
- Evaluate various C# instrumentation techniques and frameworks (e.g., Mono.Cecil, Fody, or C# Interceptors) to determine the most suitable approach for trace collection.
- Implement the chosen instrumentation strategy to successfully capture and export LINQ execution traces.
- Ensure the generated traces integrate correctly with the existing marble diagram visualizer.

Modalities:

The progress of the project should be discussed at least every three to four weeks with the advisor(s). A time schedule and a milestone plan must be set up within the first 2 weeks and discussed with the advisor(s). It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 30.04.2026.

Abstract

Language Integrated Query (LINQ) is a central feature of C# for expressing data-processing tasks in a concise and expressive way. However, beginners often find it difficult to understand the run-time behavior of LINQ queries because intermediate transformations and the resulting data flow are not directly visible.

This thesis presents an approach for visualizing LINQ query execution by instrumenting compiled C# programs at the Intermediate Language (IL) level. Using Mono.Cecil, LINQ operations in compiled assemblies are identified and instrumented with tracing functionality that records execution data for intermediate and terminal operations. The recorded trace data is serialized to JSON and imported into the graphical debugger JavaWiz for visualization. By reusing and adapting the existing JavaWiz infrastructure for the Java Stream API, the approach enables the visualization of how elements move through the individual steps of a LINQ query.

As a result, the approach supports teaching C# LINQ by allowing educators to make intermediate query transformations visible and by helping beginners understand how query results are produced.

Kurzfassung

Language Integrated Query (LINQ) ist ein zentrales Feature von C#, um Datenverarbeitungsaufgaben auf prägnante und aussagekräftige Weise auszudrücken. Anfängern fällt es jedoch oft schwer, das Laufzeitverhalten von LINQ-Abfragen zu verstehen, da die Zwischentransformationen und der daraus resultierende Datenfluss nicht direkt sichtbar sind.

Diese Arbeit stellt einen Ansatz zur Visualisierung der Ausführung von LINQ-Abfragen vor, indem kompilierte C#-Programme auf der Ebene der Intermediate Language (IL) instrumentiert werden. Mithilfe von Mono.Cecil werden LINQ-Operationen in kompilierten Assemblies identifiziert und mit einer Tracing-Funktionalität instrumentiert, die Ausführungsdaten für Zwischen- und Endoperationen aufzeichnet. Die aufgezeichneten Trace-Daten werden in JSON serialisiert und zur Visualisierung in den grafischen Debugger JavaWiz importiert. Durch die Wiederverwendung und Anpassung der bestehenden JavaWiz-Infrastruktur für die Java Stream API ermöglicht der Ansatz die Visualisierung, wie Elemente sich durch die einzelnen Schritte einer LINQ-Abfrage bewegen.

Dadurch unterstützt der Ansatz den Unterricht in C# LINQ, indem er Lehrkräften ermöglicht, Zwischenumwandlungen der Abfrage sichtbar zu machen, und Anfängern hilft zu verstehen, wie Abfrageergebnisse erzeugt werden.

Contents

Abstract	i
Kurzfassung	i
1 Introduction	1
2 Background	2
2.1 C# Language Integrated Query (LINQ)	2
2.1.1 LINQ Syntax Forms	2
2.1.2 LINQ Execution Model	3
2.2 .NET Intermediate Language (IL)	4
2.3 Mono.Cecil	4
2.4 JavaWiz	5
3 Overview	6
3.1 Purpose of the System	6
3.2 System Workflow	6
3.3 System Architecture	7
3.4 System Scope and Supported Functionality	8
4 Trace Model and Representation	9
4.1 Trace Requirements	9
4.2 Representation of LINQ Operations	9
4.3 Internal Trace Data Model	10
4.4 Run-time Generation of Trace Events	12
4.5 JSON Serialization	13
4.6 Example of a Recorded LINQ Execution Trace	14
5 Instrumentation of LINQ Queries Using Mono.Cecil	17
5.1 Instrumentation Requirements and Design Goals	17
5.2 Importing the Trace Subsystem	17
5.3 Instrumentation Approach	18
5.3.1 Detecting LINQ Operations in IL	18
5.3.2 Instrumenting Intermediate Operations	19
5.3.3 Replacing Terminal Operations	19
5.4 Example of Instrumenting a LINQ Query	20
6 Visualization of LINQ Execution in JavaWiz	23
6.1 Visualization Pipeline	23
6.2 Import and Internal Representation of LINQ Trace Data	23
6.3 Conversion into Visualization Structures	24
6.4 Example Visualization	25

7	Usage	28
7.1	General Workflow for Visualizing a LINQ Query	28
7.2	Examples	28
7.2.1	Filtering and Transforming a Collection	29
7.2.2	Flattening and Deduplicating Collections	30
8	Evaluation	32
8.1	Experimental Setup and Methodology	32
8.2	Results and Discussion	33
8.2.1	Execution Time	33
8.2.2	Trace File Size and Memory Considerations	34
8.2.3	Summary	34
9	Related Work	35
10	Limitations and Future Work	37
11	Conclusion	38
	Appendices	39
A	Non-simplified IL Example	39
	Literature	41

1 Introduction

Understanding the run-time behavior of programs is an important aspect of learning programming and software development. However, beginners often find it difficult to understand program behavior, especially when higher-level abstractions are involved. One such abstraction is C# Language Integrated Query (LINQ), which simplifies common data-processing tasks by expressing them as query pipelines. Although LINQ enables concise and expressive code, many beginners struggle to understand how data flows through the individual steps of a query, how intermediate results are produced, and how the final result relates to the original data [20, 1, 7].

Program visualization tools can support program comprehension by making run-time behavior and data transformations visible through execution traces. One example is JavaWiz, a graphical debugger designed for software development education [20]. However, LINQ query execution is still poorly supported by current visualization tools. Many debuggers and similar visualization environments show only the final result of LINQ queries, while intermediate transformations remain hidden. Moreover, tools that provide access to intermediate results are often not designed for beginners and may be difficult to use. As a result, beginners cannot easily observe how individual LINQ operators transform input data, which elements pass through filters, or how intermediate collections are constructed. This gap makes it difficult to provide intuitive explanations of how LINQ queries operate at run time. To address this limitation without requiring changes to source code, this thesis investigates a post-compilation instrumentation approach for compiled .NET assemblies. Using Mono.Cecil, compiled C# programs containing LINQ queries can be instrumented and traced so that their data flow can be visualized in JavaWiz.

This thesis makes several contributions. It presents an automated instrumentation approach for LINQ queries in compiled C# programs based on Mono.Cecil. It introduces a tracing mechanism that records the execution of LINQ method chains, including intermediate states before and after non-terminal operations. It also provides a technique for replacing terminal LINQ operations with custom tracing methods that capture final query results. In addition, a JSON-based trace format compatible with the JavaWiz visualization framework is defined. Finally, the approach is integrated into JavaWiz so that recorded traces can be visualized and the data flow through each step of a LINQ query can be examined.

Section 2 presents the theoretical foundations of the thesis, including LINQ, IL, Mono.Cecil, and JavaWiz. Section 3 then introduces the system, its workflow, architecture, and main components. Sections 4 to 6 describe the technical core of the approach. Section 4 explains trace capture and JSON serialization. Section 5 describes the instrumentation of compiled programs. Section 6 explains the transformation of the generated trace into a visualization. Section 7 demonstrates the system from the user's perspective through representative examples and Section 8 evaluates the run-time overhead introduced by tracing. Section 9 compares the approach with existing techniques and tools, while Section 10 discusses the current limitations of the system and outlines possible improvements. Finally, Section 11 summarizes the results of the thesis and reflects on its contribution.

2 Background

This section introduces the theoretical and technical foundations required for the thesis. It first discusses C# LINQ, focusing on its syntax forms and execution model. Then it introduces the Intermediate Language (IL) used in compiled assemblies. Thereafter, Mono.Cecil is presented as the library used for IL instrumentation. Finally, JavaWiz is discussed, which already supports the visualization of Java stream pipelines and is extended in this thesis to support C# LINQ visualization.

2.1 C# Language Integrated Query (LINQ)

Language Integrated Query (LINQ) is a set of technologies in the C# programming language that allows developers to query and transform data directly within C#. LINQ provides a unified abstraction for querying and transforming data from different sources, including in-memory data and remote data (e.g., relational databases). However, for this thesis, we focus on in-memory LINQ queries, that is, *LINQ to Objects*. Typically, LINQ queries are expressed as chains of operations that either process elements one by one or consume the full input before producing a result. A LINQ query consists of three distinct actions. The first step is obtaining the data source. The second is specifying the query, that is, the sequence of operations, which does not yet trigger execution. The final step is executing the query. Each operation takes a sequence as input and produces either a transformed sequence or a final aggregated result. This distinction between operations will be explained in more detail in Section 2.1.2. Common LINQ operations include filtering (**Where**), projection (**Select**), ordering (**OrderBy**), grouping (**GroupBy**), and materialization (**ToList**) [14, 16, 19, 5].

To use LINQ to Objects, the in-memory data must be of a type that implements `IEnumerable<T>`, since these methods are implemented as extension methods defined in the `System.Linq` namespace [14, 16, 17].

2.1.1 LINQ Syntax Forms

LINQ provides two syntactic forms for defining queries. The first is called *method syntax*, and the second is the *declarative query syntax* [19]. The two syntax forms are described below.

Method syntax: Method syntax expresses queries using chained method calls. It directly reflects the underlying LINQ method calls and, therefore, does not require compiler translation into method call form. The main advantage of method syntax is that it supports the full range of LINQ methods, whereas query syntax is more limited [19]. An example of this syntax is illustrated in Listing 1.

```
1 IEnumerable<int> methodSyntax = numbers
2   .Where(num => num < 5)
3   .OrderBy(num => num);
```

Listing 1: Simple example of LINQ method syntax.

Query Syntax: Query syntax resembles SQL-like declarative queries. Query syntax was designed to be more readable and easier to understand, especially for beginners, which is also the main advantage of using query syntax instead of method syntax. However, there is a drawback, because some operations do not exist in query syntax and as a result, queries containing these operations, such as `Count` or `Max`, must be expressed in method syntax [16, 19, 5]. An example of this syntax is illustrated in Listing 2.

```
1 IEnumerable<int> querySyntax =  
2     from num in numbers  
3     where num < 5  
4     orderby num  
5     select num;
```

Listing 2: Simple example of LINQ query syntax.

Equivalence of Both Syntax Forms: Although the two syntax forms look different, the C# compiler translates query syntax into method calls during compilation, which is the same as using method syntax. As a result, both forms produce equivalent compiled code and show no performance difference [19, 5]. This property is especially important for this thesis because the instrumentation can focus on compiled method calls rather than the source-level syntax.

2.1.2 LINQ Execution Model

As already mentioned, LINQ queries form pipelines of operations. These operations either produce a transformed sequence, in which case they are *intermediate operations*, or a final result, in which case they are *terminal operations*. This section explains the types of operations in more detail.

Intermediate Operations: Intermediate operations transform sequences, but do not immediately execute the query. They are lazily evaluated and are only executed when the sequence is consumed, for example, by a terminal operation or a `foreach` loop. This is called *deferred execution*. To allow LINQ operation chaining, they return another `IEnumerable<T>`. Intermediate operations can further be divided into *streaming* and *nonstreaming* operations. Streaming operations process each element as it is read and yield it before the next element is processed. Examples include `Select` and `Where`. Nonstreaming operations read the full input data source before performing the operation. This is required for some methods, for example `OrderBy` or `GroupBy` [14].

Terminal Operations: In contrast to intermediate operations, terminal operations trigger the execution of the full query pipeline. In essence, these operations cause all preceding intermediate operations to be executed. They typically also produce a final result, instead of another `IEnumerable<T>` and, if used, are the last operation in the LINQ chain. In the context of this thesis, terminal operations are particularly important because they represent the point at which a query pipeline is evaluated and both intermediate and terminal results become available [14].

Many LINQ to Objects queries rely on deferred execution. This behavior is important for this thesis because it makes the run-time behavior of LINQ pipelines less directly visible to learners and motivates the need for tracing and visualization [14].

For a broader overview, Microsoft provides a classification table of LINQ operations [14].

2.2 .NET Intermediate Language (IL)

The discussion now shifts from source-level LINQ to its compiled representation. Programs written in C# are not compiled directly into machine code. Instead, they are compiled into Intermediate Language (IL), also called Common Intermediate Language (CIL), which is stored in a .NET assembly and executed by the .NET Runtime (CLR). This process is similar to Java, where source code is compiled into bytecode and executed by the Java Virtual Machine (JVM), which translates it to machine code at run time. When the program runs, the CLR uses Just-In-Time (JIT) compilation to translate IL instructions into machine code that can be executed by the CPU. Because IL preserves the structure of methods and instructions inside the assembly, it can be inspected and modified after compilation. This makes it possible to insert additional instructions for tracing, so the instrumentation used in this thesis operates on IL rather than on the original C# source code [4].

2.3 Mono.Cecil

Mono.Cecil is a library for inspecting, modifying, and generating .NET assemblies in the ECMA CIL format. It provides access to type and method definitions, as well as individual IL instructions. These capabilities make it possible to read a compiled assembly, analyze its structure, insert additional instructions, modify existing instructions, and write the transformed assembly back to disk. Mono.Cecil therefore serves as the foundation for injecting additional functionality, such as run-time tracing, into compiled C# programs [3, 21].

Rationale for Using Mono.Cecil: Since C# programs may use different syntactic forms for LINQ, the instrumentation approach used in the thesis must work independently of the source-level representation. Source code instrumentation would require handling these syntactic variations explicitly. In contrast, IL-level instrumentation operates on the compiled representation of the program, where query syntax has already been translated into method call form. This allows the system to detect LINQ method chains, insert tracing instructions, and capture the run-time data flow regardless of how the query was written. Other approaches, such as compile-time weaving with Fody or C# Interceptors, were also considered. However, they were deemed less suitable for the post-compilation instrumentation adopted in this thesis. A detailed comparison is provided in Section 9. Mono.Cecil therefore provides a suitable mechanism for instrumenting LINQ queries in compiled .NET assemblies without requiring users to modify their source code.

Advantage of IL-Level Instrumentation: The IL-level instrumentation approach provides multiple advantages. First, no access to the original source code is required. In addition, the approach works for any compiled C# program by providing the assembly to be instrumented.

Also, differences in source-level syntax do not affect the instrumentation, since both forms generate the same method calls. Finally, the resulting traces accurately reflect the actual executed LINQ queries. Overall, this approach provides a syntax- and source code-independent way to instrument compiled C# programs and to capture the actual run-time behavior of LINQ queries.

2.4 JavaWiz

The final foundation for this thesis is JavaWiz, which provides the visualization environment for the recorded trace data. JavaWiz is a program visualization tool designed to help beginners understand program execution. The tool visualizes the run-time behavior and data flow of Java programs based on execution traces. These traces describe the program state at each execution step, including variables and their values as well as object relationships. This information is presented through multiple interactive visualizations that allow users to step through program execution. This approach helps users observe how data structures evolve during run time [20].

From the Java Stream API to C# LINQ: Originally, JavaWiz only supported the visualization of Java stream pipelines, which are conceptually similar to LINQ queries. The Java Stream API and LINQ share several characteristics. Both operate on collections, use pipelines of transformation operations, and distinguish between intermediate and terminal operations. Because of these similarities, LINQ query chains can be visualized using concepts similar to those used for Java streams. This is advantageous because, by creating an appropriate execution trace format, LINQ operations can be mapped to the visualization model used by JavaWiz. This approach enables the visualization of intermediate transformations, element-wise data flow and final results of LINQ queries.

3 Overview

This section provides a high-level overview of the system. It first introduces its purpose, then outlines the workflow, presents the system architecture and main components, and finally summarizes the system scope and supported functionality.

3.1 Purpose of the System

The purpose of the system is to support beginners in understanding the execution of LINQ queries in C# programs. To achieve this, compiled programs are instrumented so that execution traces of LINQ operations are recorded during run time and imported into JavaWiz for interactive visualization. This allows users to inspect intermediate results and data flow through a LINQ query.

3.2 System Workflow

The workflow from user code to visualization begins with a C# program containing LINQ queries and ends with the visualization of the recorded trace in JavaWiz.

First, the program is compiled into a .NET assembly. The instrumentation component then analyzes the assembly, detects LINQ method chains, and uses Mono.Cecil to insert or replace tracing instructions. The original assembly is then overwritten with an instrumented version. When the instrumented assembly is executed, the run-time tracing component records trace events and writes them to a JSON file. This JSON trace is then imported into JavaWiz, where it is parsed and transformed into visualization structures. This enables JavaWiz to visualize the data flow through each step of the LINQ pipeline.

Figure 1 provides an early preview of the resulting JavaWiz visualization. It shows how elements are represented as they pass through the individual operations of a LINQ query pipeline. The visualization process and its internal representation are explained in detail in Section 6.

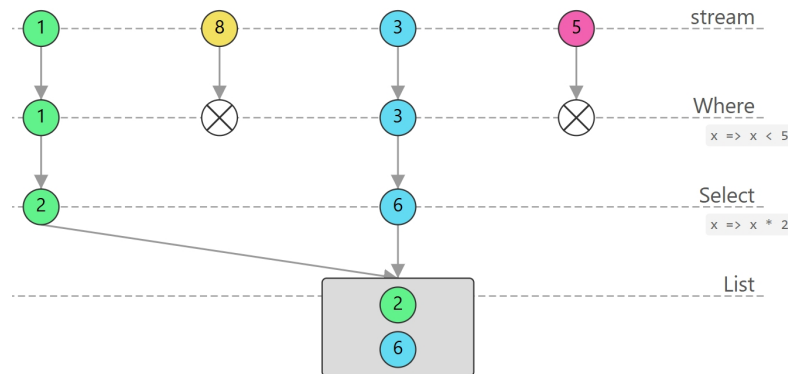


Figure 1: Preview of a LINQ query execution visualized in JavaWiz.

3.3 System Architecture

Building on the previous section, the main components of the system and their responsibilities are described in more detail. Figure 2 illustrates the system architecture and data flow of the LINQ visualization system. The numbered components shown in the figure are explained below.

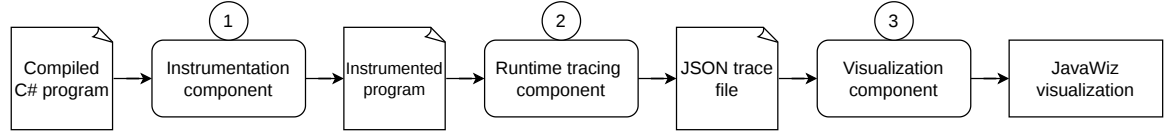


Figure 2: System architecture and data flow of the LINQ visualization system, from the compiled C# program through instrumentation and run-time tracing to the final visualization in JavaWiz.

① **Instrumentation component:** This component is responsible for modifying compiled assemblies and is the first processing component in the system pipeline. It relies on Mono.Cecil for manipulating the compiled assembly, which contains the program instructions in Intermediate Language (IL). Its features include:

- Loading compiled .NET assemblies.
- Analyzing method bodies.
- Identifying LINQ method chains.
- Inserting trace calls around intermediate LINQ operations.
- Replacing terminal LINQ operations.
- Overwriting the assembly with an instrumented version.

② **Run-time tracing component:** This component is responsible for collecting trace data from LINQ queries during program execution. Trace collection is performed automatically when the instrumented user assembly is executed. Its features include:

- Recording trace events for LINQ operations executed by the instrumented program.
- Capturing relevant execution data for each operation.
- Storing the collected trace data during run time.
- Serializing the recorded trace data into a JSON file which can be imported into JavaWiz.

③ **Visualization component:** This component is responsible for loading and parsing the JSON file produced by the run-time tracing component and generating the visualization structures required by JavaWiz. Its features include:

- Importing the JSON trace file.
- Converting the trace data into internal Kotlin objects representing the trace.
- Creating the visualization structures required by JavaWiz from the converted trace data.

Overall, the system enables interactive visualization of LINQ query execution.

3.4 System Scope and Supported Functionality

This section defines the scope and supported functionality of the system. The system supports the detection of LINQ method chains and the instrumentation of compiled C# assemblies containing LINQ queries. It can insert tracing logic before and after intermediate LINQ operations and replace terminal LINQ operations with custom tracing methods while preserving the semantics of the original terminal call. In addition, it generates JSON execution traces that describe the behavior of LINQ queries and enables their visualization in JavaWiz by generating the required visualization structures.

This high-level overview provides the background needed to understand the system as a whole and serves as the basis for the next three sections, which discuss the individual subsystems in more detail.

4 Trace Model and Representation

This section introduces the trace model and its representation. The trace model defines the run-time information captured by the system and the trace method calls inserted by the instrumentation subsystem described in Section 5. It first presents the trace requirements and then explains the conceptual representation of LINQ operations within the trace model and the internal structure of the trace model. Finally, it describes the run-time generation of trace events, serialization to JSON, and provides an example illustrating how the tracing subsystem works in practice.

4.1 Trace Requirements

To visualize LINQ execution, the trace must capture specific run-time information. In essence, it must represent the execution of LINQ query pipelines so that each individual operation in a chain can be identified, and the processing of elements through these operations can be reconstructed. It must also capture the data flow between operations.

For each operation, the trace must record:

- execution order,
- operation type,
- trace direction,
- element value,
- element type,
- parent element relationship,
- and, for some operations, the operation arguments.

This information must be compatible with the JavaWiz visualization requirements so that the execution structure can be reconstructed correctly. In addition, the complete trace should be generated automatically during program execution without requiring additional user input or modification of the source code.

The next section introduces the conceptual representation of LINQ queries in the trace model.

4.2 Representation of LINQ Operations

LINQ to Objects queries consist of chains of operations applied to collections that implement the `IEnumerable` interface. As discussed in Section 2.1.2, these operations can be categorized into intermediate operations and terminal operations. Intermediate operations transform a sequence and return another sequence, but do not trigger query execution. In contrast, terminal operations produce a final result and therefore do trigger query execution [14].

In the trace model, the input and output of an intermediate operation may be represented by distinct trace events. This is possible because intermediate operations transform one sequence into another on an element-by-element basis. However, there are special cases in which no output event is generated. One example is a filtering operation such as `Where`, in which some elements

may not pass the filter. Terminal operations cannot be traced in exactly the same way, since they produce a final result rather than a per-element output sequence. They nevertheless also produce trace events, but their trace event generation follows a different strategy; this strategy is explained in more detail in Section 4.4. For the first LINQ operation in a chain, the trace model introduces a special start event for each input element. This is necessary because JavaWiz visualizations are based on Java stream pipelines, in which a stream begins with an explicit **stream** stage.

This representation is illustrated in Figure 3. Thus, intermediate operations may generate separate input and output trace events for an element, filtered-out elements may produce only an input trace event, and terminal operations are represented by an **END** trace event.

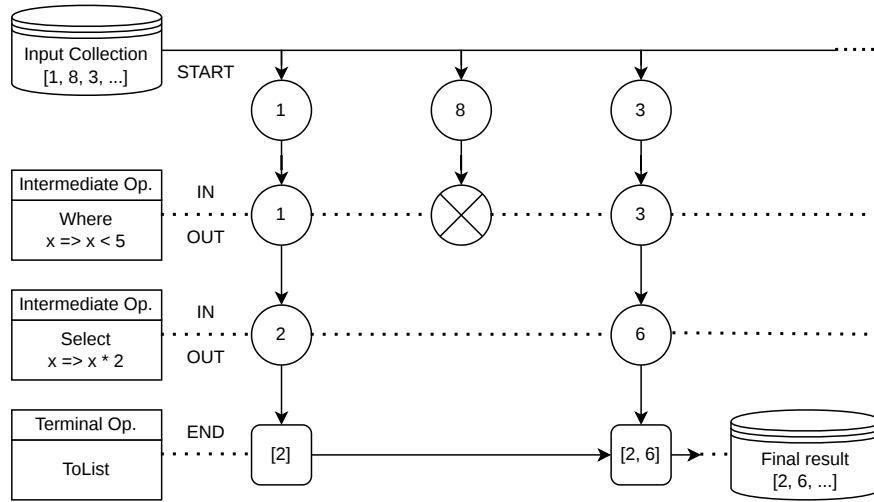


Figure 3: Object-oriented representation of LINQ operations in the trace model.

Finally, operations are recorded in the order in which they are executed so that the LINQ chain can later be reconstructed correctly.

4.3 Internal Trace Data Model

Based on this representation, trace events must also be stored in a suitable internal data model. The trace data is stored as a sequence of events that belong to a query execution, generated by the tracing subsystem. Internally, these events are represented by the **StreamOperationValue** class. Figure 4 illustrates this structure using two example trace entries: an output **Select** event and a terminal **ToList** event. The numbered field descriptions below correspond to the numbered markers in Figure 4.

- ① **seq**: Indicates the order of the event for visualization-relevant trace entries. In general, it increments for every new event. However, if the element is an input, the value is always zero, since these entries are not directly visible in the visualization.
- ② **type**: This field stores the name of the LINQ method.
- ③ **direction**: Specifies whether the trace event represents an input event, an output event, or a special event such as the start or end of a LINQ chain.

- ④ **operationID**: Since similar LINQ methods can be chained (for example multiple **Where**-clauses), each method in the chain is assigned a unique operation ID. In essence, it describes the method ID in the LINQ chain.
- ⑤ **elementID**: Every element has a unique ID to track every element throughout the LINQ chain.
- ⑥ **parentIDs**: This field contains a list of all elements that are predecessors of the current event to allow linking of trace events in the visualization.
- ⑦ **valueType**: Stores the data type of the current element value. It is used to display the elements correctly in the JavaWiz visualization, since different visualizations are used depending on the value type.
- ⑧ **value**: Stores the value of the current element. It is used to display the value of the element in JavaWiz.
- ⑨ **param**: Some methods, such as filtering operations (e.g. **Where**), provide arguments that can be displayed in JavaWiz to get more detailed information about a LINQ method in the visualization.
- ⑩ **rawValueType**: Some operations use a different **valueType** in JavaWiz than in the user's C# code, for example, **GroupBy** uses a **Dictionary/Map** in JavaWiz to display the element, whereas in C# it is an **IGrouping**. This field stores the real value type of the current element instead of the display type.
- ⑪ **rawValue**: This field is analogous to **rawValueType**, but stores the raw value rather than its type. For example, user-defined class objects are converted into a **string** representation for visualization in JavaWiz and stored in **value**. The corresponding **rawValue** field retains the original object instance so that it can still be used in subsequent tracing steps.

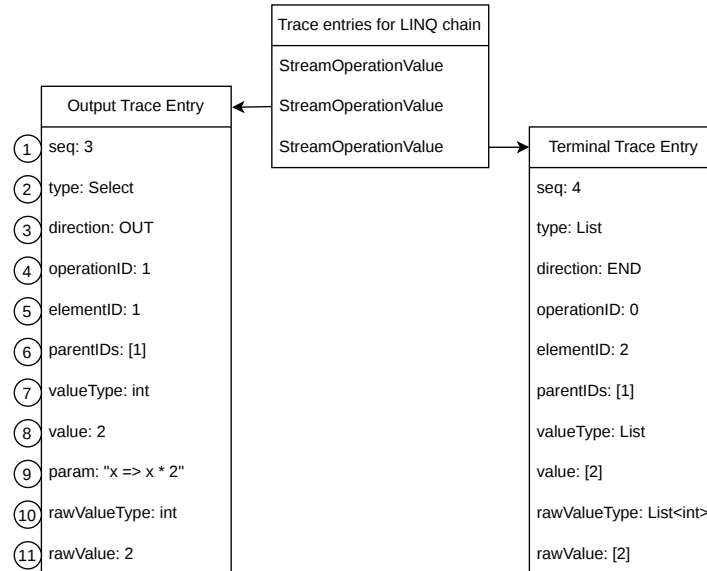


Figure 4: Conceptual structure of the internal trace model.

This model is based on the current JavaWiz stream trace model and, therefore, supports reconstruction and visualization of the data flow between operations. With this approach, all elements can be tracked as they move through the query pipeline.

Based on this data structure, the following section explains how trace events are generated at run time.

4.4 Run-time Generation of Trace Events

To generate trace events, run-time methods are used that record the information required for later visualization. Different run-time methods are used depending on the detected LINQ operation. As already mentioned in Section 4.2, the main distinction is between intermediate and terminal operations. This section explains the two approaches used in the run-time tracing component and discusses the special **START** trace event.

Intermediate operation: To trace intermediate operations, a method named `TraceLinq` is created. This method is inserted before and after every intermediate operation in order to record the required trace information. The same method can be used for both input and output tracing because `TraceLinq` has an argument named `tracePhase`, which distinguishes between the two main tracing phases (IN and OUT). For ordering operations, such as `OrderBy` or `ThenBy`, a special method called `TraceOrderedLinq` is used, since these methods use the `IOrderedEnumerable` interface instead of `IEnumerable`. This method creates a wrapper for `IOrderedEnumerable` to allow tracing. However, this wrapper then calls `TraceLinq` internally to reuse the existing trace functionality.

Terminal operation: As mentioned in Section 4.2, this approach does not apply directly to terminal operations. However, it is possible to add per-element tracing functionality by replacing the terminal method with a custom tracing-aware implementation. These dedicated tracing-aware implementations are available for most terminal operations, including `ToList`, `ToDictionary`, `Count`, `First`, and `Last`. This replacement mechanism is explained in more detail in Section 5.3.3. To represent terminal operations, each input element is added to an internal collection, and the original terminal method is repeatedly executed on this growing collection to produce distinct trace events. This approach records the evolving result during execution. In this way, terminal operations can also be represented in the trace model, even though their behavior differs from that of intermediate operations.

If no suitable replacement method is available, a fallback based on the intermediate-operation approach is used to provide limited tracing of terminal operations. However, there is a difference since the output produces a final result instead of another `IEnumerable` sequence. For the output, a method called `TraceTerminal` is created to trace the final output of the terminal operation. This fallback is less detailed than the replacement approach because it does not capture how the result evolves for each input element. It is currently used for operations with many overloads. Examples include `Sum`, `Min`, or `Max`.

Special case, first operation in a LINQ chain: A special case occurs when tracing the first LINQ operation in a chain. In this case, an additional stream-start trace event is created for each input element before the actual operation is traced. These events represent the initial

stream source in the visualization and are required because JavaWiz is based on Java stream pipelines, in which execution begins with an explicit `stream` stage. This special handling can apply regardless of whether the first LINQ operation is intermediate or terminal.

Although the methods differ in their implementation, they all serve the same basic purpose. Each trace method captures the current operation, records the input and output elements, and stores the information in a trace structure. The resulting trace events are then written to a trace file for each LINQ chain, as explained in the next section.

4.5 JSON Serialization

During execution, trace events are generated and stored in memory. After the required trace information has been recorded, it must be exported for later visualization in JavaWiz. For this purpose, JSON is used as the serialization format, since it is human-readable, easy to parse, and widely supported.

Exporting Trace Data to a JSON File: In the current implementation, collected trace data is written to a file after the execution of each LINQ query pipeline so that the latest trace data is available for visualization. This serialization process converts the internal trace structures, that is, the list of `StreamOperationValue` objects associated with each LINQ chain, into JSON objects. Internally, trace data is stored as a `Dictionary<int, List<StreamOperationValue>>`, where each LINQ chain is identified by a unique ID. Since this internal C# model is already compatible with the structure expected by JavaWiz, it can be serialized directly using the `Newtonsoft.Json` package without requiring an additional transformation step in the trace subsystem. The resulting JSON file represents query-pipeline execution in a format that can later be visualized.

Mapping Between Internal Data Model and JSON Representation: First, the JSON representation closely mirrors the structure of the internal trace data model explained earlier. That means that nearly each attribute of a `StreamOperationValue` is written directly as a JSON field. However, as discussed in Section 4.3, two additional fields are not serialized. These fields are `rawValueType` and `rawValue`. They are ignored because they are only used in the run-time tracing components to preserve the original value and type information, whereas the serialized `value` and `valueType` fields may represent transformed values that are then used for visualization purposes. This preserves the semantic structure of the internal model and avoids the need for additional transformation during serialization. Because of this direct mapping, the JSON structure remains predictable and allows the visualization tool to easily reconstruct the execution trace.

The following section illustrates the serialization result using a recorded LINQ execution.

4.6 Example of a Recorded LINQ Execution Trace

To illustrate how the tracing subsystem records the execution of a LINQ query, consider the running example shown in Listing 3.

```
List<int> numbers = [1, 8, 3, 5];  
numbers.Where(x => x < 5).Select(x => x * 2).ToList();  
          ①                ②                ③
```

Listing 3: LINQ query used as the running example for trace generation and visualization.

Conceptual Execution of the Query: In this example, we can see that the input collection consists of a sequence of integers. The query contains two intermediate operations and a terminal operation.

① **Where** is the first intermediate LINQ operation. The operation filters elements according to the given predicate [13]. In this example, each input element of the input sequence is evaluated against the predicate $x < 5$. Elements that do not satisfy the condition are removed and therefore not further evaluated. The result after the filtering step is the following: [1, 3].

② **Select** is the second intermediate LINQ operation. This operation projects each element of a sequence into a new form [10]. That means that each remaining element that passed the filter is transformed using the expression: $x * 2$. Therefore, the result after this step is: [2, 6].

③ **ToList** is the terminal operation that executes the whole query. The operation produces the final collection that contains the remaining elements after the intermediate operations [12]. In our example, the final result is [2, 6]. Note that this operation collects the `IEnumerable<int>` sequence into a `List<int>`.

Trace Events Generated During Execution: The trace events generated during execution can now be examined in relation to this query structure. For the first element, the **Where** operation produces three events: a **START** event when a new object is first processed in the query, an **IN** event when the object enters the **Where**, and an **OUT** event if the object "survives" the **Where**. The **Select** operation then produces two additional events, one for input and one for output. Finally, the terminal **ToList** operation creates an **END** event for each object added to the list. Together, these events make it possible to reconstruct the query execution.

JSON Representation of the Trace: After execution, the program creates a JSON file containing the query trace. The JSON output shown in Listing 4 corresponds to the trace events recorded by the instrumented program. For example, the events associated with value 1 show that the element passes through the **Where** operator, is transformed into 2 by **Select**, and is subsequently added to the result list.

```

1  {
2    "seq": 1, "type": "stream", "direction": "START", "operationID": 3, "elementID": 1,
3    "parentIDs": [1], "valuetype": "int", "value": 1, "param": ""
4  },
5  {
6    "seq": 0, "type": "Where", "direction": "IN", "operationID": 2, "elementID": 1,
7    "parentIDs": [1], "valuetype": "int", "value": 1, "param": "x => x < 5"
8  },
9  {
10   "seq": 2, "type": "Where", "direction": "OUT", "operationID": 2, "elementID": 1,
11   "parentIDs": [1], "valuetype": "int", "value": 1, "param": "x => x < 5"
12 },
13 {
14   "seq": 0, "type": "Select", "direction": "IN", "operationID": 1, "elementID": 1,
15   "parentIDs": [1], "valuetype": "int", "value": 1, "param": "x => x * 2"
16 },
17 {
18   "seq": 3, "type": "Select", "direction": "OUT", "operationID": 1, "elementID": 1,
19   "parentIDs": [1], "valuetype": "int", "value": 2, "param": "x => x * 2"
20 },
21 {
22   "seq": 4, "type": "List", "direction": "END", "operationID": 0, "elementID": 2,
23   "parentIDs": [1], "valuetype": "List", "value": [2], "param": ""
24 },
25 {
26   "seq": 5, "type": "stream", "direction": "START", "operationID": 3, "elementID": 3,
27   "parentIDs": [3], "valuetype": "int", "value": 8, "param": ""
28 },
29 {
30   "seq": 0, "type": "Where", "direction": "IN", "operationID": 2, "elementID": 3,
31   "parentIDs": [3], "valuetype": "int", "value": 8, "param": "x => x < 5"
32 },
33 {
34   "seq": 6, "type": "stream", "direction": "START", "operationID": 3, "elementID": 4,
35   "parentIDs": [4], "valuetype": "int", "value": 3, "param": ""
36 },
37 {
38   "seq": 0, "type": "Where", "direction": "IN", "operationID": 2, "elementID": 4,
39   "parentIDs": [4], "valuetype": "int", "value": 3, "param": "x => x < 5"
40 },
41 {
42   "seq": 7, "type": "Where", "direction": "OUT", "operationID": 2, "elementID": 4,
43   "parentIDs": [4], "valuetype": "int", "value": 3, "param": "x => x < 5"
44 },
45 {
46   "seq": 0, "type": "Select", "direction": "IN", "operationID": 1, "elementID": 4,
47   "parentIDs": [4], "valuetype": "int", "value": 3, "param": "x => x * 2"
48 },
49 {
50   "seq": 8, "type": "Select", "direction": "OUT", "operationID": 1, "elementID": 4,
51   "parentIDs": [4], "valuetype": "int", "value": 6, "param": "x => x * 2"
52 },
53 {
54   "seq": 9, "type": "List", "direction": "END", "operationID": 0, "elementID": 2,
55   "parentIDs": [1, 4], "valuetype": "List", "value": [2, 6], "param": ""
56 },
57 {
58   "seq": 10, "type": "stream", "direction": "START", "operationID": 3, "elementID": 5,
59   "parentIDs": [5], "valuetype": "int", "value": 5, "param": ""
60 },
61 {
62   "seq": 0, "type": "Where", "direction": "IN", "operationID": 2, "elementID": 5,
63   "parentIDs": [5], "valuetype": "int", "value": 5, "param": "x => x < 5"
64 }

```

Listing 4: JSON output of the recorded LINQ query trace.

The fields shown in Listing 4 correspond directly to the internal trace model introduced in Section 4.3. The `seq` field increases for all events other than `IN` events, for which its value is 0. The `type` field reflects the operation name, while the `direction` field specifies which phase of the traced operation the event belongs to, such as `START`, `IN`, `OUT`, or `END`. The `operationID` identifies the position of the operation within the LINQ chain, and the `elementID` identifies the current element. The `parentIDs` field stores the elements that are connected to the trace event. The `value` and `valuetype` fields are used to display the element information correctly in JavaWiz. Finally, the `param` field stores the lambda expression associated with the operation.

Together, these fields preserve the order of operations and the lineage of elements across the query pipeline.

The trace model and run-time tracing methods defined in this section establish the requirements for instrumentation. To generate the required events, the instrumentation must distinguish between intermediate and terminal operations and insert or replace method calls accordingly. The following section explains how this is achieved at the IL level using Mono.Cecil.

5 Instrumentation of LINQ Queries Using Mono.Cecil

Building on the trace model and its recording requirements, this chapter explains how compiled programs are instrumented to produce a modified assembly that generates the required trace data at run time. In the context of this thesis, the compiled program is typically provided as a *dynamic-link library (DLL)*, that is, a .NET assembly stored in a .dll file. The instrumentation is implemented using Mono.Cecil, which allows compiled .NET assemblies to be inspected and modified. The chapter focuses on how LINQ operations are identified in IL and how tracing behavior is inserted into compiled code.

5.1 Instrumentation Requirements and Design Goals

Several requirements must be taken into account when instrumenting the target assembly. First, the instrumentation must operate on compiled programs without requiring access to the source code. Second, tracing must be performed automatically for LINQ queries without requiring user intervention. A central requirement is that the insertion and replacement of trace method calls must not alter the original program semantics, since the goal is to observe LINQ execution rather than change program behavior. Finally, because the system is intended for novice programmers, it should require as little additional setup as possible and support both query and method syntax.

5.2 Importing the Trace Subsystem

This section explains how the tracing subsystem described in Section 4 is imported into the compiled assembly so that it can be used during instrumentation. Two main approaches were considered for making the tracing methods available in the target assembly. The first approach is to add the trace library as a separate dependency and reference it through the user's *.deps.json* file. The alternative is to merge the trace DLL directly into the user's DLL using a custom merging tool such as ILRepack. After evaluating both options, the second approach was selected. This approach was chosen for several reasons:

- It is easier for novice developers to execute the instrumented program and use the system, since there are no additional DLLs.
- Normally, additional DLLs have to be included in the *.deps.json* file. By direct import via custom merging tools, there is no need to change the file and the user does not have to provide it in addition to the DLL.
- The approach allows for simpler instrumentation, because references to trace methods do not need to be imported separately in Mono.Cecil.

To implement this approach, ILRepack is used to merge the DLL with the trace subsystem before instrumentation. This is done by calling ILRepack via the command line to merge the DLL with the trace subsystem.

```

1 static void MergeWithTool(string main, string ext, string output)
2 {
3     var psi = new ProcessStartInfo
4     {
5         FileName = Path.Combine(AppContext.BaseDirectory, "ILRepack.exe")
6     };
7     psi.ArgumentList.Add($"/out:{output}"); ①
8     psi.ArgumentList.Add(main); ②
9     psi.ArgumentList.Add(ext); ③
10    psi.ArgumentList.Add("Newtonsoft.Json.dll"); ④
11    psi.ArgumentList.Add("/internalize");
12
13    using var p = Process.Start(psi)!;
14    p.WaitForExit();
15 }

```

Listing 5: ILRepack invocation for merging the trace subsystem into the target DLL.

Listing 5 provides the C# code used to invoke ILRepack as an external process.

- ① The `/out` argument specifies the path of the merged output assembly that will contain both the user's program and the trace subsystem.
- ② The `main` argument refers to the compiled user DLL that should be instrumented.
- ③ The `ext` argument refers to the trace subsystem DLL, which contains the tracing methods that are merged into the user assembly.
- ④ The `Newtonsoft.Json` assembly must also be included in the merge because the trace subsystem uses it to serialize trace data to JSON during program execution.

By combining these three assemblies into a single output DLL, the system ensures that tracing functionality is available during instrumentation and execution without requiring separate DLLs to be distributed.

5.3 Instrumentation Approach

With the trace subsystem available in the target assembly, the instrumentation can insert the required tracing behavior. In IL, LINQ queries appear as chained method calls, which makes it possible to analyze method bodies and instructions and detect LINQ operations automatically. After a LINQ operation has been identified, the instrumentation determines whether it is an intermediate or a terminal operation. Intermediate operations are instrumented by inserting tracing calls before and after the original method call, whereas terminal operations are replaced with tracing-aware implementations. As a result, the compiled program is transformed into a version that produces trace data during execution. The following sections explain the detection and transformation steps in more detail.

5.3.1 Detecting LINQ Operations in IL

As discussed earlier, LINQ operations appear as method calls in the compiled IL code. Consequently, the instrumentation does not need to distinguish between query syntax and method syntax at the source level. For the in-memory LINQ operations considered in this thesis, the

relevant calls are defined in `System.Linq.Enumerable` [17]. The instrumentation component identifies these calls by inspecting method bodies and IL instructions.

To classify a method call as a relevant LINQ operation, several conditions must be satisfied. First, the declaring type of the method must be `System.Linq.Enumerable`. Second, the method must be static and implemented as an extension method, since standard LINQ to Objects operations are provided as extension methods on `IEnumerable<T>`. These conditions make it possible to distinguish LINQ operations from unrelated method calls in the compiled code.

In addition to identifying LINQ operations in general, instrumentation must also distinguish between intermediate and terminal operations. This is done by inspecting the return type of the method. If the return type is `IEnumerable`-like, the operation is treated as intermediate. Otherwise, if the return type itself is not an enumerable interface, it is treated as terminal.

Using these checks, the instrumentation can identify relevant in-memory LINQ operations in compiled code and distinguish between intermediate and terminal pipeline stages. Section 5.3.2 and Section 5.3.3 describe how the detected intermediate and terminal operations are instrumented.

5.3.2 Instrumenting Intermediate Operations

When an intermediate LINQ operation is detected, instrumentation is performed by inserting tracing calls before and after the operation. This design captures both the input and output needed for the visualization. In addition, the arguments required by the tracing methods must be inserted before the tracing method call. Listing 6 provides a high-level example of a LINQ operation instrumentation. However, for readability, the listing omits the additional IL instructions used to load the arguments for the inserted trace calls. A more detailed example follows in Section 5.4.

```
// Original IL
call System.Linq.Enumerable::Where

// Instrumented IL
call TraceLinq.TraceLinqExtensions::TraceLinq ①
call System.Linq.Enumerable::Where
call TraceLinq.TraceLinqExtensions::TraceLinq ①
```

Listing 6: Simplified example of instrumentation for an intermediate LINQ `Where` operation.

① The inserted calls record the input and output of the intermediate LINQ operation. These calls invoke the run-time trace methods described in Section 4.

This makes it possible to record intermediate pipeline states without changing the original query, because the trace methods pass the original value to the next operation. It allows the system to observe how elements change across each operation.

Terminal operations require a different instrumentation strategy, which is described in the next section.

5.3.3 Replacing Terminal Operations

In contrast to intermediate operations, terminal operations trigger the execution of the LINQ pipeline and produce the final result. They collect all elements and then return the final output.

Therefore, simply inserting tracing around them does not provide sufficient detail. To support detailed tracing of terminal LINQ to Objects queries over `IEnumerable`, the original method call is replaced with a custom tracing-aware implementation when a dedicated replacement method is available. Methods or overloads for which no dedicated replacement exists fall back to the non-terminal tracing approach. In that case, the standard `TraceLinq` method is inserted before the terminal call, and a `TraceTerminal` call is inserted afterward. However, because the fallback can only trace the final output of the terminal method, it does not preserve per-element tracing for the terminal operation. This design allows the final query result to be recorded as part of the trace while preserving the original program behavior.

```
// Original IL
call System.Linq.Enumerable::ToList

// Instrumented IL when a dedicated replacement is available
call TraceLinq.TraceLinqExtensions::ToListTrace ①

// Instrumented IL when a dedicated replacement is not available
call TraceLinq.TraceLinqExtensions::TraceLinq
call System.Linq.Enumerable::ToList
call TraceLinq.TraceLinqExtensions::TraceTerminal ②
```

Listing 7: Simplified example of instrumentation for a terminal LINQ `ToList` operation.

① As we can see, the terminal operation is replaced by the custom tracing-aware `ToListTrace` implementation.

② If no suitable replacement is available, the original call is not replaced. Instead, the approach of Section 5.3.2 is used to provide limited terminal tracing.

This instrumentation approach ensures that the trace captures the complete result of the query pipeline.

5.4 Example of Instrumenting a LINQ Query

Building on the theoretical description, the instrumentation process is illustrated using the running example from Section 4.6 and is shown again in Listing 8. Recall that the query contains a filtering step (`Where`), a transformation step (`Select`), and a terminal collection step (`ToList`).

```
numbers.Where(x => x < 5).Select(x => x * 2).ToList()
```

Listing 8: Running example LINQ query used to illustrate the instrumentation process.

To focus on the important parts of the instrumentation, Listing 9 and Listing 10 omit IL instructions that are not essential for the example.

```
1 call System.Linq.Enumerable::Where
2 call System.Linq.Enumerable::Select
3 call System.Linq.Enumerable::ToList
```

Listing 9: The simplified compiled representation of the example query.

As shown in Listing 9, the IL only contains the call instructions for the query operations. The instrumentation transforms the IL into the representation presented in Listing 10. For readability, lower-level compiler-generated details are omitted. The complete, non-simplified IL is provided in Appendix A.

```

1  stloc.s predicate
2  ldstr "Where"
3  ldc.i4.0
4  ldstr "x => x < 5"
5  call TraceLinq.TraceLinqExtensions::TraceLinq ①
6
7  ldloc.s predicate
8  call System.Linq.Enumerable::Where ②
9
10 ldstr "Where"
11 ldc.i4.1
12 ldstr "x => x < 5"
13 call TraceLinq.TraceLinqExtensions::TraceLinq ③
14
15 // For readability, the argument-loading instructions are omitted
16 // for the following operation, since the same pattern has
17 // already been shown for Where
18 call TraceLinq.TraceLinqExtensions::TraceLinq
19 call System.Linq.Enumerable::Select
20 call TraceLinq.TraceLinqExtensions::TraceLinq ④
21
22 call TraceLinq.TraceLinqExtensions::ToListTrace ⑤

```

Listing 10: Simplified instrumented representation of the example query.

① These instructions are inserted before the **Where** call. This **TraceLinq** call corresponds to the before (IN) phase, i.e., capturing the input of the **Where** operation. The inserted trace call is preceded by instructions that load the arguments required for the trace method. This includes operation name, the trace phase (e.g., `ldc.i4.0` indicates an IN-phase), and in some cases the predicate expression. The same general argument-loading pattern is used for the **Select** operation, but is omitted in the listing for readability.

② Then the predicate is loaded onto the stack again and the original **Where** LINQ operation is invoked.

③ This **TraceLinq** call corresponds to the after (OUT) phase, now capturing the output. As shown in the instrumented code, both trace call instructions invoke the same trace method. This is because both before- and after-tracing create similar **StreamOperationValue** objects, while the distinction between the two cases is handled inside the trace method.

④ The same pattern is then applied to the next intermediate operation, **Select**.

⑤ Finally, the **ToList** terminal operation is replaced by a custom tracing-aware implementation **ToListTrace**. As shown in the IL code, no additional IL changes are required beyond replacing the method call itself, since the tracing-aware method has the same signature as the original terminal operation.

Figure 5 shows the instrumentation log for the running example. In addition to the standard query, the example is also executed as a parallel query to illustrate the extra feedback produced by the tool. The log reports the merge of the trace subsystem into the user assembly, the detection of supported LINQ operations, and unsupported query forms that are recognized but not instrumented, such as Parallel LINQ. This feedback is useful for debugging and makes the current scope limitations of the tool visible in practice.

```
Starting dll merge!
----- IL Repack Arguments -----
/out:BscThesisExamples.dll BscThesisExamples.dll TraceLinqExtensions.dll Newtonsoft.Json.dll /internalize
-----

Finished dll merge!
Starting Instrumentation!
[BscThesisExamples] BScThesisExamples.Program::Main -> System.Linq.Enumerable::Where()
[BscThesisExamples] BScThesisExamples.Program::Main -> System.Linq.Enumerable::Select()
[BscThesisExamples] BScThesisExamples.Program::Main -> System.Linq.Enumerable::ToList()
Parallel LINQ methods are not supported, skipping method: System.Linq.ParallelEnumerable::AsParallel()
Parallel LINQ methods are not supported, skipping method: System.Linq.ParallelEnumerable::Where()
Parallel LINQ methods are not supported, skipping method: System.Linq.ParallelEnumerable::Select()
Parallel LINQ methods are not supported, skipping method: System.Linq.ParallelEnumerable::ToList()
Finished Instrumentation!
```

Figure 5: Instrumentation log for the running example, showing the merge of the trace subsystem, the detection of LINQ operations, and the recognition of unsupported Parallel LINQ calls.

The resulting trace data provides the basis for the JavaWiz visualization described in the following section.

6 Visualization of LINQ Execution in JavaWiz

This section explains how the generated LINQ trace is transformed and then visualized in JavaWiz.

6.1 Visualization Pipeline

Figure 6 illustrates the LINQ trace visualization pipeline from the recorded JSON trace to the final JavaWiz visualization. The numbered descriptions below correspond to the markers in Figure 6.

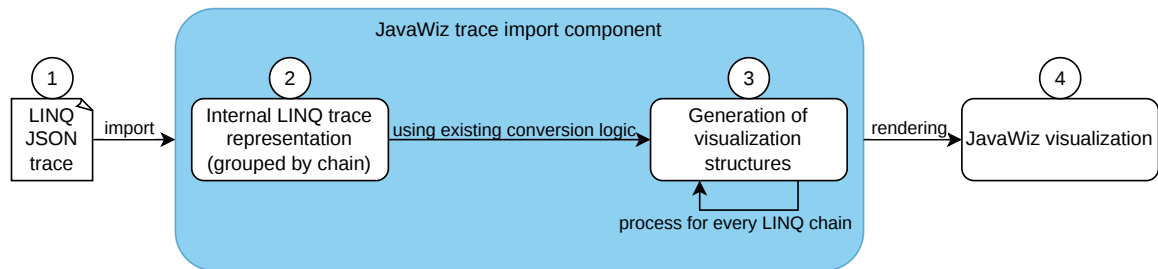


Figure 6: Visualization pipeline that transforms a recorded JSON trace into an internal LINQ trace representation and finally into the JavaWiz visualization structures.

① First, the instrumented program produces the JSON trace file after execution. This file contains all the necessary information for visualization. The user must manually import it into JavaWiz.

② After the user imports the trace file, JavaWiz parses it into an internal representation, one per chain of LINQ statements.

③ Then, each LINQ chain is processed separately using the existing JavaWiz stream visualization logic. This step produces the set of visualization objects used by the visualization component.

④ Finally, the visualization objects generated for each LINQ chain are rendered by JavaWiz.

Section 6.2 and Section 6.3 describe the individual stages of the visualization subsystem in detail.

6.2 Import and Internal Representation of LINQ Trace Data

In step ② of the visualization pipeline, the visualization import loads and parses the JSON trace and converts it into its internal representation.

Loading the trace: To support C# in JavaWiz, a dialog was added that allows the user to import the JSON trace. After submission, the trace is sent to the backend, where the entries are parsed into Kotlin objects representing the recorded LINQ execution steps. To allow JavaWiz to handle multiple queries in one trace, the data is grouped by the LINQ chain ID.

Structure of the loaded trace: After loading the trace data, it is represented internally as a map from LINQ chain IDs to lists of `StreamOperationValue` objects. As discussed earlier in Section 4.3, the C# `StreamOperationValue` class has a corresponding Kotlin representation. Since the Kotlin class serves the same purpose, the classes are nearly identical. Only the two fields `rawValue` and `rawValueType` are omitted because they are not used for visualization and are therefore no longer needed after program execution. This internal Kotlin representation preserves the information required for later conversion and rendering of the trace.

6.3 Conversion into Visualization Structures

The internal trace representation is subsequently converted into the visualization structures used by JavaWiz. This corresponds to step ③ of the visualization pipeline.

Reuse of the Existing JavaWiz Conversion Logic: JavaWiz already provides a function for converting Java stream trace data to visualization objects. This function can be reused to convert the C# trace data into the same kind of visualization objects. Listing 11 illustrates the conversion algorithm.

```

1 private fun convertLinqJsonToStreamTrace(streamOperationValues:
    MutableMap<Int, MutableList<StreamOperationValue>>):
    List<StreamVisualizationInfo> {
2     val tracer = StreamOperationTracer() ①
3     tracer.streamtrace = streamOperationValues
4     val values = streamOperationValues.map { (key, _) ->
5         tracer.actualStreamID = key ②
6         val temp = tracer.collectAndTransformStreamOperationValues() ③
7         StreamVisualizationInfo(temp.marbles.toMutableList(), ④
8             temp.links.toMutableList(),
9             temp.operationLines.toMutableMap(),
10            temp.lastX, temp.lastOpId)
11    }
12    return values
13 }

```

Listing 11: Conversion function for transforming trace data into visualization information.

① First, a `StreamOperationTracer` instance is created. It stores the trace data passed to the function and offers the conversion logic used to create the visualization objects.

② Internally, the `collectAndTransformStreamOperationValues` function converts a LINQ chain based on the `actualStreamID` field value. So, to allow conversion of all LINQ chains, the value must be changed for every LINQ chain in the trace.

③ Next, the core conversion function `collectAndTransformStreamOperationValues` is invoked, and the resulting visualization data is assigned to a temporary variable.

④ Finally, for every LINQ chain, a `StreamVisualizationInfo` object is created that essentially contains the data returned by the conversion function. These objects are then used to render each LINQ chain.

Reusing the existing JavaWiz conversion and visualization structures avoids the need for a separate rendering pipeline and reduces maintenance effort because future changes to the visualization model can be handled within a shared conversion path.

Generation of Visualization Data: The `StreamVisualizationInfo` class was introduced earlier. Its main visualization structures used by JavaWiz during rendering are the following:

- **Marbles** represent visible values or objects in the visualization.
- **Links** represent relationships between elements across operations. As discussed earlier, the `parentIDs` field is used to keep track of these relationships.
- **OperationLines** represent the visual rows of the pipeline. This is done by tracking the type of operation and also the `operationID` during tracing.

These structures form the basis for the final visualization in JavaWiz. Based on the generated visualization structures, JavaWiz displays the LINQ pipeline as a sequence of visual stages. Users can therefore observe which elements pass or fail filter operations, how values are transformed, how intermediate results evolve, and also how the final result is produced. This helps make the run-time behavior of LINQ queries more accessible to beginners.

6.4 Example Visualization

To illustrate the visualization component in practice, the running example from Section 4.6 is used to show how the recorded trace data is transformed and visualized in JavaWiz.

```
numbers.Where(x => x < 5).Select(x => x * 2).ToList()
```

Listing 12: Running example LINQ query used to illustrate the visualization process in JavaWiz.

As a reminder, the example query shown again in Listing 12 consists of a filter step (**Where**), a transformation step (**Select**), and a terminal collection step (**ToList**).

After executing the instrumented program containing the query, the JSON trace file is created, which must then be imported manually by the user. Internally, the trace entries are represented as a map. Listing 13 shows the resulting `map` structure for element 1 after JavaWiz loads the submitted JSON trace already shown in Section 4.6.

```
1 0 = [  
2 StreamOperationValue(seq=1, type=stream, direction=START, value=1, ...),  
3 StreamOperationValue(seq=0, type=Where, direction=IN, value=1, ...),  
4 StreamOperationValue(seq=2, type=Where, direction=OUT, value=1, ...),  
5 StreamOperationValue(seq=0, type=Select, direction=IN, value=1, ...),  
6 StreamOperationValue(seq=3, type=Select, direction=OUT, value=2, ...),  
7 StreamOperationValue(seq=4, type=List, direction=END, value=[2], ...),  
8 ...  
9 ]
```

Listing 13: Simplified imported `StreamOperationValue` map excerpt.

For readability, the excerpt focuses on the entries associated with element 1 and omits structurally similar entries for other elements.

As we can see in Listing 13, the JSON data was reconstructed into Kotlin `StreamOperationValue` objects and is nearly identical to the objects in the C# trace component (without the two additional fields mentioned above). JavaWiz then applies its existing conversion logic to this internal representation. The data after conversion is now shown in Listing 14.

```

1 ① marbles = [ ...,
2    {"id": "1.2", "y": 100, "label": "1", "operationID": 2,
3      "type": "Where", "color": "#61f28b", "step": 2, ...},
4    {"id": "1.1", "y": 200, "label": "2", "operationID": 1,
5      "type": "Select", "color": "#61f28b", "step": 3, ...}, ...
6  ]
7 ② links = [ ..., {"source": "1.2", "target": "1.1", "step": 3}, ... ]
8 ③ operationLines = [ ..., {"type": "Where", "y": 100}, ... ]

```

Listing 14: Simplified visualization data excerpt.

For readability, the excerpt focuses on the entries associated with element 1 with the `Where` and `Select` operations and omits structurally similar entries for other elements and operations.

As shown in Listing 14, the `StreamOperationValues` are converted into three substructures: `marbles`, `links`, and `operationLines`.

① In the `marbles` data, the `id` is composed of the `elementId` and the `operationID`, separated by a dot. The `y` value determines the vertical position of the marble in the visualization. The `label` field stores the displayed value of the object. The `color` field defines the color used for rendering. And the `step` field specifies the visualization step at which the marble becomes visible.

② In the `links` data, the `source` and `target` fields identify the marbles that are directly connected.

③ In the `operationLines` data, the `type` field stores the operation name, while the `y` value determines the vertical position at which the operation should be rendered in the visualization.

Figure 7 shows the resulting visualization in JavaWiz. In this example, we can see that elements 1 and 3 passed the filtering step (`Where`), then they were transformed, i.e., multiplied by two in the `Select` operation to 2 and 6, and finally collected into a list: `[2, 6]`.

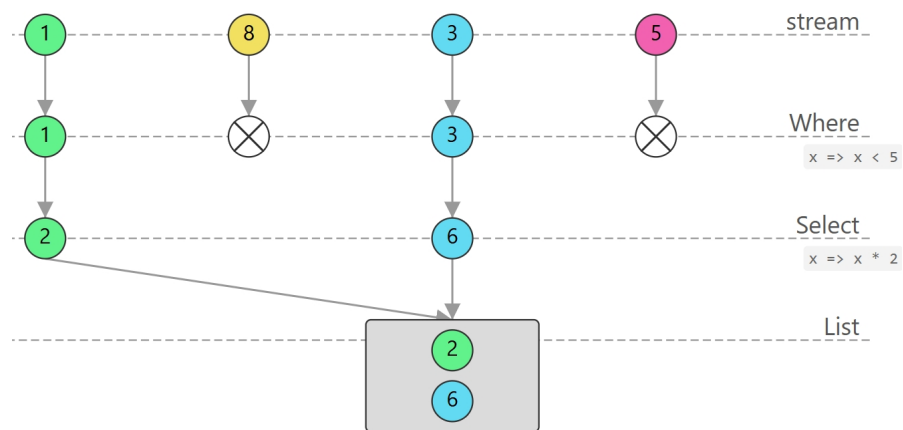


Figure 7: Final JavaWiz visualization of the running example, showing that elements 1 and 3 pass the filter, are transformed to 2 and 6, and are collected into the final list.

In conclusion, the resulting visualization aims to help users better understand LINQ queries by making their execution visible from start to finish.

7 Usage

Having presented the technical realization of the system, this section now considers its usage from the user's perspective. It first explains the general workflow for visualizing a LINQ query and then illustrates this workflow with two example queries.

7.1 General Workflow for Visualizing a LINQ Query

Before presenting concrete examples, the general workflow for visualizing a LINQ query is outlined. Figure 8 illustrates the workflow from writing a C# program to viewing the resulting visualization in JavaWiz. The following numbered steps correspond to the markers in Figure 8 and describe the workflow in more detail.

- ① The user writes a C# program containing one or more LINQ queries.
- ② The program is compiled into a .NET assembly using the standard C# compiler.
- ③ The compiled assembly is processed by the instrumentation component, which inserts the trace functionality required for LINQ query execution.
- ④ The original program assembly is replaced by an instrumented version.
- ⑤ During execution, trace events for LINQ operations are recorded.
- ⑥ The trace component then writes the recorded trace data to a JSON trace file.
- ⑦ The visualization component groups the trace by LINQ chain, parses it, and converts it into the data structures required for visualization.
- ⑧ The resulting JavaWiz visualization shows the query pipeline and the data flow through each operation.

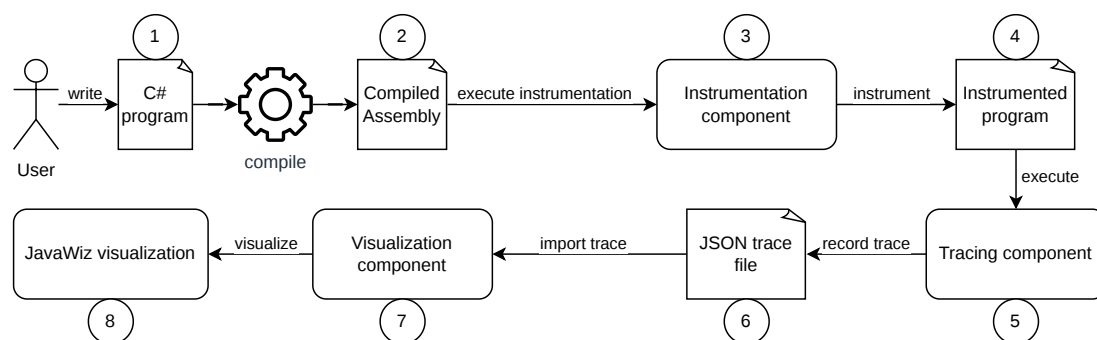


Figure 8: Workflow diagram for visualizing a LINQ query.

7.2 Examples

Two examples are used to show how the system works in practice. The first example is simpler, whereas the second one is more complex since it demonstrates transformations with nested collections.

7.2.1 Filtering and Transforming a Collection

This example has already been used to explain the implementation. Here, the focus shifts to the user's perspective in order to show how the system is used in practice.

```
List<int> numbers = [1, 8, 3, 5];
numbers.Where(x => x < 5).Select(x => x * 2).ToList();
```

Listing 15: LINQ query that filters and transforms a collection.

Recall that the query shown in Listing 15 keeps elements smaller than five, multiplies the remaining elements by two, and collects the result into a list of integers.

First, the user compiles the program and runs the instrumentation tool on the resulting DLL by specifying the DLL path as a command-line argument. Figure 9 shows a console excerpt from the instrumentation step. The output lists the methods that were detected and instrumented with tracing functionality (`Where`, `Select`, and `ToList`). For readability, the excerpt omits the DLL merge already discussed in Section 5.4.

```
Starting Instrumentation!
[BscThesisExamples] BscThesisExamples.Program::Main -> System.Linq.Enumerable::Where()
[BscThesisExamples] BscThesisExamples.Program::Main -> System.Linq.Enumerable::Select()
[BscThesisExamples] BscThesisExamples.Program::Main -> System.Linq.Enumerable::ToList()
Finished Instrumentation!
```

Figure 9: Console output of the instrumentation step, showing that the LINQ operations `Where`, `Select`, and `ToList` were detected and instrumented with tracing functionality.

At this point, the user's DLL has been overwritten with a trace-ready version that can then be executed, for example, by running the command `"dotnet .\<DLL name>"` if the *runtimeconfig.json* file is available at the same path.

After execution, the program produces a JSON trace file that can be imported into JavaWiz. To import the trace, the user opens the dialog shown in Figure 10 from the JavaWiz stream view by pressing the `C#` button.

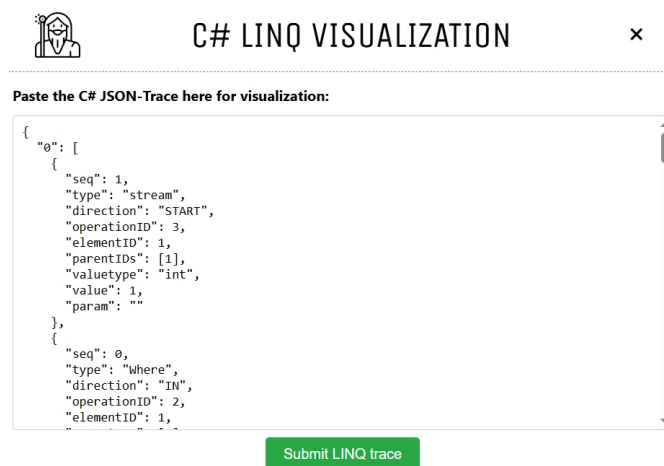


Figure 10: JavaWiz import dialog for submitting the generated JSON trace.

After the *Submit LINQ trace* button is pressed, the trace data is sent to the backend and parsed into the visualization structures used by the frontend. Figure 11 shows the visualization at step six, where element 1 passes the filter, while element 8 does not. Figure 12 then shows the visualization at the final step, where the full query pipeline, intermediate results, and the final output ([2, 6]) are visible.

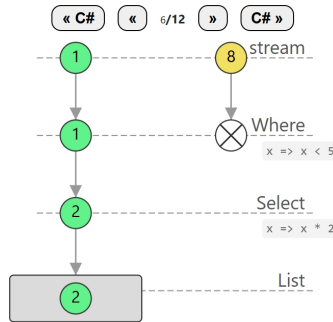


Figure 11: JavaWiz visualization of the running example at step six, showing that element 1 passes the filter while element 8 is removed by the *Where* operation.

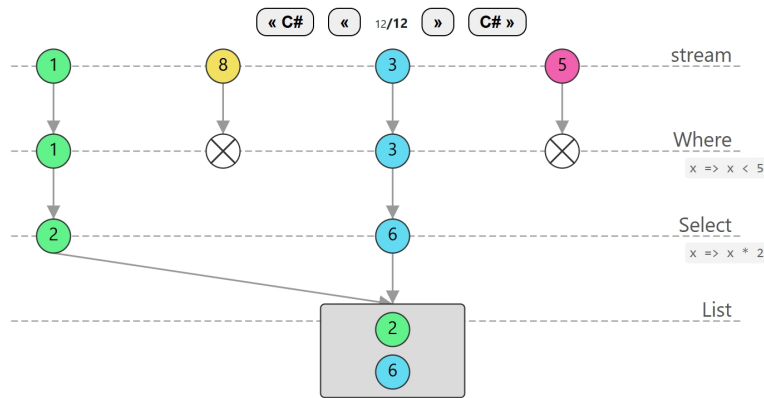


Figure 12: Final JavaWiz visualization of the running example, showing the complete query pipeline, the intermediate results, and the final output [2, 6].

With this visualization approach, the user can observe how each element moves through the query pipeline.

7.2.2 Flattening and Deduplicating Collections

This example is more complex because it focuses on *SelectMany*, which flattens multiple sequences into a single sequence.

```
List<List<int>> lists = [ [4, 2], [2, 1] ];
lists.SelectMany(x => x).Distinct().Count();
```

Listing 16: LINQ query that flattens collections and then deduplicates the resulting collection.

As seen in Listing 16, this query consists of **SelectMany**, **Distinct**, and **Count**. **SelectMany** projects each element of the lists to an **IEnumerable** and flattens the sequences into one sequence [11]. JavaWiz must therefore create visualization objects for the elements produced from the nested lists. Then, **Distinct** removes duplicate elements, and lastly, the terminal operation **Count** counts all remaining elements of the sequence [8, 9].

Since the workflow up to the final visualization is essentially the same as in Section 7.2.1, this example focuses mainly on the resulting visualizations. Figure 13 illustrates how the query processes the first list. **SelectMany** projects each element of the first list to an **IEnumerable**. Thereafter, **Distinct** removes duplicate elements. The first list does not contain any duplicate elements, so both elements are retained. Last, the elements are counted and, since both elements were retained by the **Distinct** step, the count is 2.

Figure 14 shows the final visualization of the complete query pipeline. **SelectMany** flattens both lists into a single sequence. In the **Distinct** operation row, the visualization shows that a duplicate element is detected. In this case, the element with value 2 is removed by the **Distinct** operation and is therefore not processed further. Finally, the element with value 1 passes the **Distinct** operation and is counted, yielding the final result of 3.

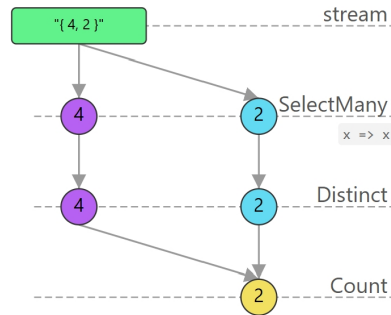


Figure 13: JavaWiz visualization of the flattening example at step 7, showing how the first nested list is flattened and counted without removing any duplicates.

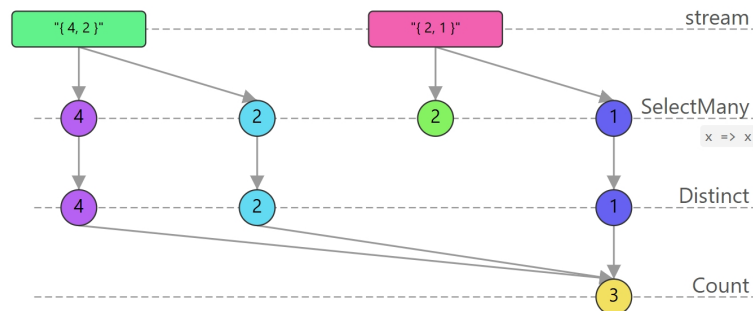


Figure 14: Final JavaWiz visualization of the flattening example, showing that the duplicate value 2 is removed by **Distinct** and the final count is 3.

Together, these examples show how the system can visualize both basic and more complex LINQ pipelines and make their transformations visible.

8 Evaluation

Although the system is designed for novice programmers, its run-time overhead must still be assessed to determine whether it remains acceptable for typical use cases. This section examines how tracing affects execution time and trace size, how these overheads scale with increasing input size, and how query structure and terminal operations influence the results.

8.1 Experimental Setup and Methodology

To evaluate the performance impact of the proposed system, two representative LINQ query patterns were executed under controlled conditions. The queries are based on the examples introduced in Section 7 and were chosen to study how tracing overhead is affected by query structure and terminal LINQ operations.

For each query, three versions of the program were measured: The *baseline* version without instrumentation, the *instrumented (no I/O)* version with tracing but omitting JSON serialization and file writing, and the *instrumented (I/O)* version, with both tracing and trace persistence. This separates tracing overhead from serialization and file-writing overhead.

All measurements were performed on the same machine and under the same run-time conditions using .NET 10.0. For each configuration, the average execution time over 100 runs was recorded using the `Stopwatch` class in C#. Each run was started in a fresh process, so cold-start effects, including JIT-compilation, were included deliberately, since this reflects the intended usage scenario of the system more closely. For the I/O configuration, the trace file was overwritten after each execution.

Each query was evaluated with input sizes of 10, 100 and 1000 elements. These sizes reflect typical educational examples while still showing scaling behavior.

The query set was designed to examine two factors. First, the second query extends the first by introducing `SelectMany`, allowing the influence of an additional intermediate operation while keeping the total number of processed primitive elements constant. Second, for both query patterns, the terminal operation was varied between `ToList` and `Count` to compare collection-producing and scalar terminal operations. This is relevant because terminal operations affect both the amount of generated trace data and, when trace persistence is enabled, the cost of serialization and file writing. The evaluated queries are shown in Listing 17.

```
numbers.Where(x => x % 2 == 0).Select(x => x * 2)
    .<ToList() | Count()>;
lists.SelectMany(x => x).Where(x => x % 2 == 0).Select(x => x * 2)
    .<ToList() | Count()>;
```

Listing 17: Evaluated LINQ queries with alternative terminal operations.

To compare baseline and instrumented execution, three metrics were used: average execution time in milliseconds (ms), absolute overhead (Δ_{abs}) in ms, and relative overhead (Δ_{rel}) in percent (%). The relative overhead was calculated as follows: $\Delta_{rel} = \frac{T_{instrumented} - T_{baseline}}{T_{baseline}} \times 100$

In addition to execution time, the size of the generated traces was evaluated. For each query and input size, the number of generated trace events, the result size, and the size of the serialized JSON trace file were recorded. To examine the impact of the terminal operation type, the trace size was measured for both `ToList` and `Count`.

8.2 Results and Discussion

This section discusses the results of the evaluated queries. First, the performance overhead is analyzed, followed by the size of the generated trace files.

8.2.1 Execution Time

Input	Base (ms)	Instr. no I/O (ms)	Δ_{abs} (ms)	Δ_{rel} (%)	Instr. I/O (ms)	Δ_{abs} (ms)	Δ_{rel} (%)
numbers.Where(x => x % 2 == 0).Select(x => x * 2).ToList()							
10	1.338	15.613	14.275	1066.89	82.175	80.837	6041.63
100	1.897	15.738	13.841	729.63	83.772	81.873	4316.03
1000	1.921	18.463	16.542	861.11	133.609	131.688	6855.18
numbers.Where(x => x % 2 == 0).Select(x => x * 2).Count()							
10	1.342	16.432	15.090	1124.44	84.114	82.772	6167.81
100	1.385	16.430	15.045	1086.28	86.841	85.456	6170.11
1000	1.380	19.118	17.738	1285.36	119.058	117.678	8527.39

Table 1: Execution times and overhead of the base LINQ query.

Input	Base (ms)	Instr. no I/O (ms)	Δ_{abs} (ms)	Δ_{rel} (%)	Instr. I/O (ms)	Δ_{abs} (ms)	Δ_{rel} (%)
lists.SelectMany(x => x).Where(x => x % 2 == 0).Select(x => x * 2).ToList()							
10	1.954	20.702	18.748	959.47	88.462	86.508	4427.23
100	2.555	20.278	17.723	693.66	90.225	87.670	3431.31
1000	2.684	23.765	21.081	785.43	140.905	138.221	5149.81
lists.SelectMany(x => x).Where(x => x % 2 == 0).Select(x => x * 2).Count()							
10	1.888	20.198	18.310	969.81	88.956	87.068	4611.65
100	1.887	20.562	18.675	989.67	91.384	89.497	4742.82
1000	2.007	23.756	21.749	1083.66	127.819	125.812	6268.66

Table 2: Execution times and overhead of the LINQ query with **SelectMany**.

The results in Table 1 and Table 2 show that the baseline execution time remains nearly constant across all input sizes, indicating that for the considered datasets, constant overhead dominates the run-time. Without trace persistence, instrumentation introduces a noticeable but acceptable absolute overhead. In this configuration, the overhead is primarily influenced by the structure of the query. Within the intended educational input range, the effect of input size is negligible and becomes noticeable only for substantially larger inputs. In particular, the inclusion of additional operations, such as **SelectMany**, increases execution time by several milliseconds, while the choice of terminal operation has only a minor impact.

When trace data is serialized and written to disk, execution time increases significantly, reaching up to about 140 ms. This overhead is dominated by JSON serialization and file writing, which add a substantial fixed cost. As a result, small input sizes exhibit similar run-times, whereas larger inputs lead to higher execution times due to the growing amount of trace data. In contrast, the choice of terminal operation becomes a more important factor when trace persistence is enabled. Materializing operations such as **ToList** generate more trace data and therefore produce higher overhead, whereas aggregating operations such as **Count** result in lower overhead.

8.2.2 Trace File Size and Memory Considerations

Input elements	Result size	Generated trace events	JSON trace size (kB)	
			ToList()	Count()
numbers.Where(x => x % 2 == 0).Select(x => x * 2)				
10	8	52	7.402	7.277
100	51	404	65.505	60.555
1000	485	3940	1445.815	1015.808
lists.SelectMany(x => x).Where(x => x % 2 == 0).Select(x => x * 2)				
10	3	40	5.799	5.778
100	55	488	80.762	75.179
1000	476	4566	1541.984	1121.061

Table 3: Sizes of the generated JSON trace files for the two evaluated LINQ queries.

The results in Table 3 show that the trace file size increases considerably with the number of processed elements. This growth is mainly influenced by the number of generated trace events and the amount of data stored in each event. Operations that reduce or expand the number of processed elements, such as `Where` and `SelectMany`, directly affect the number of generated trace events and therefore the resulting trace size.

The terminal operation also noticeably impacts the trace size. Traces generated for `ToList` are consistently larger than those generated for `Count`, and this difference becomes more pronounced for larger inputs. This is caused by the way terminal operations are represented in the trace. Operations that return a scalar result, such as `Count`, generate `END` trace events whose `value` field contains only a scalar value. In contrast, collection-producing operations such as `ToList` generate events whose `value` field contains the accumulated result collection. As the result grows, previously collected values are serialized repeatedly, causing the storage overhead of `ToList` to increase more strongly with the number of result elements.

For the evaluated LINQ queries, trace sizes range from approximately 5–7 kB for 10 input elements to about 1.0–1.5 MB for 1000 input elements.

Memory Overhead: Trace events are stored in memory as `StreamOperationValue` objects before serialization. The number of generated trace events scales approximately with $O(N \times M)$ where N is the number of elements and M is the number of pipeline stages. However, the memory requirement also depends on the size of the data stored in trace events. In particular, collection-producing terminal operations may store increasingly large accumulated result collections. Consequently, memory usage can increase considerably for large inputs or complex queries. However, this overhead is expected to remain manageable for the intended use cases.

8.2.3 Summary

Overall, the results indicate that instrumentation introduces moderate overhead without I/O and a more substantial overhead when trace persistence is enabled. Trace file size and memory consumption also increase with the number of generated events, which may lead to an `OutOfMemoryException` in the worst case. Nevertheless, for the intended educational input sizes, both execution time and trace size remain within an acceptable range, making the approach suitable for its intended purpose.

9 Related Work

This section discusses related work relevant to the instrumentation and visualization approach presented in this thesis. The related work can be divided into two main areas: approaches for instrumenting program behavior in .NET, and approaches for visualizing pipeline-based query execution. In the first area, Fody and C# Interceptors are considered as alternative instrumentation mechanisms and compared with the Mono.Cecil-based approach used in this thesis. In the second area, the existing visualization of Java stream pipelines in JavaWiz is discussed as the conceptual basis for the visualization of C# LINQ queries.

Fody: Fody is a compile-time weaving framework for the .NET ecosystem. Its main purpose is to support assembly manipulation during the build process. Common use cases include logging or code generation [2]. Fody is relevant to this thesis because it supports IL-level modification during the build process and therefore appears to be a possible basis for the instrumentation. However, several reasons make direct use of Mono.Cecil more suitable in this context.

The proposed approach is intended to work independently of the original project's build configuration and source code. Since Fody is primarily designed as a build-integrated weaving framework and would require a custom weaver as well as project-level configuration, it would add integration overhead without providing capabilities beyond those already available through direct use of Mono.Cecil, which Fody itself uses internally. Therefore, using Fody would increase complexity without providing a clear advantage for the post-compilation tracing task. Furthermore, although Fody provides add-ins for common weaving tasks, no suitable add-in was available for this thesis. Using Fody would therefore still have required a custom weaver and additional build integration, increasing complexity. Fody was therefore not selected as the instrumentation mechanism.

C# Interceptors: C# Interceptors, added in C# 12, are another relevant alternative because they allow specific method calls to be replaced with alternative implementations and may therefore appear promising for tracing LINQ operations. However, they are not well suited to the goals of the proposed approach [18].

First, Interceptors are still experimental, which makes them unsuitable for a robust instrumentation approach, since their design may change or the feature may even be removed in future releases. Second, they are intended to modify code at the source code level rather than to instrument already compiled programs. Third, they typically require explicit annotations or configuration, which conflicts with the goal of instrumenting programs without requiring source code modification. Finally, because Interceptors operate at the source code level, they do not provide the syntax-independent post-compilation instrumentation required here. For these reasons, interceptors were considered unsuitable for the thesis [18, 6].

Instrumentation of Java Stream Pipelines in JavaWiz: In addition to instrumentation-related work, this thesis is also related to existing work on visualizing pipeline-based query execution. In this context, the visualization of Java stream pipelines already implemented in JavaWiz is especially relevant. This existing functionality served as the basis for adapting the visualization concept to C# LINQ.

The two approaches are conceptually similar because both the Java Stream API and LINQ are based on pipelines of intermediate and terminal operations applied to collections. In both cases, trace data must represent the flow of elements through a sequence of operations and must distinguish between intermediate transformations and final results. This similarity makes it possible to map LINQ query execution to visualization concepts that were originally developed for Java stream pipelines.

At the same time, there are important differences in the instrumentation process. In Java, trace functionality for streams was added without third-party tools directly, while in C# a separate instrumentation approach had to be developed using Mono.Cecil. Another difference is that Java streams are typically created through explicit calls such as `stream()` or `Stream.of()`, which makes stream detection more direct. In C#, LINQ queries are expressed as chains of method calls over `IEnumerable<T>`, which requires a different detection strategy at the IL level. In addition, Java distinguishes between several stream types, such as `IntStream` and `LongStream`, which must be handled separately, while C# LINQ does not require such a distinction.

Overall, the existing approach to instrumenting and visualizing Java stream pipelines provided an important foundation. However, adapting the concept to C# LINQ required a different instrumentation strategy and several adjustments to account for the differences between the two ecosystems.

Summary: The related approaches discussed show that both instrumentation and pipeline-based visualization techniques already exist but differ in their suitability for the goals of the proposed approach. Fody and C# Interceptors are relevant alternatives, yet neither is as well suited to syntax-independent post-compilation instrumentation as direct use of Mono.Cecil. At the same time, the existing visualization of Java stream pipelines in JavaWiz demonstrates that pipeline-based data flow visualization is effective in a related setting and provides the conceptual foundation for the C# LINQ visualization developed in this thesis.

10 Limitations and Future Work

Although the system already supports a wide range of C# LINQ queries, several limitations remain and suggest directions for future work.

First, the current implementation provides limited data type support. Currently, basic data types and simple collections already work well with JavaWiz. However, complex object types may not always be represented clearly in the visualization. In particular, object types such as intermediate dictionaries are currently limited in their functionality. In addition, some object types, such as user-defined classes, depend on run-time memory references that are not directly accessible through the proposed tracing approach. Although this limitation can be partly mitigated by using alternative visualization objects, these only provide simplified representations rather than the custom layouts. Based on this, future work could focus on supporting more detailed visualization of complex data structures in addition to adding new visualization objects.

Another limitation is that tracing is currently restricted to LINQ to Objects, so the system supports only queries operating on `IEnumerable<T>` sequences. However, LINQ also supports other execution models that are currently not supported, such as `IQueryable<T>` or `Parallel LINQ (PLINQ)`. In the case of `IQueryable<T>`, queries are typically executed by external providers, such as database engines. These queries are translated into another query language (e.g. SQL), rather than executed as .NET code [15]. As a result, IL-level instrumentation cannot capture their execution behavior because these queries are not executed as regular method calls. Furthermore, PLINQ is also currently not supported because it introduces concurrent execution of query operations across multiple threads. This is problematic because it may lead to non-deterministic execution order, concurrent data processing, and interleaved trace events. Future work can therefore focus on the implementation of new tracing mechanisms for `IQueryable<T>` and PLINQ.

Furthermore, JavaWiz integration is limited and currently only supported by manual JSON import. This reduces usability, because the process is not seamless and introduces an additional manual step. The current workflow requires the user to run the instrumented program, obtain the generated JSON trace file, and load it manually into the visualization tool. Although the process is reliable, it unfortunately does not provide a fully seamless user experience. Future improvements could include direct communication between the tracing system and the visualization tool, integration through a dedicated plugin or extension, or direct C# integration into JavaWiz. Such improvements would enhance usability, especially in educational settings.

Finally, LINQ provides a large number of operations and method overloads. However, due to the scope of the thesis, the system was primarily tested with representative subsets of common LINQ operators. As a result, some specific LINQ queries and operators may still produce incomplete or incorrect trace data. Future work could extend the testing to include additional LINQ operators and complex query pipelines. This would help ensure more robust behavior across a broader portion of the LINQ API.

Overall, the system already provides a robust basis for tracing LINQ queries, while future work can improve visualization support, strengthen JavaWiz integration, and extend coverage to additional execution models and LINQ operators.

11 Conclusion

LINQ queries provide a concise and expressive way to formulate data-processing tasks in C# programs. However, their run-time behavior is often difficult for beginners to understand because intermediate transformations and the resulting data flow are not directly visible. In particular, it is difficult to observe how elements move through the individual steps of a query and how the final result is derived from the original input. This creates a need for tools that can visualize the data flow within LINQ queries.

This thesis presented an IL-level instrumentation approach for compiled C# programs containing LINQ queries. The approach records execution traces of queries with intermediate and terminal LINQ operations, serializes the recorded trace data to JSON, and imports it into JavaWiz for visualization. This is achieved by adapting the existing visualization infrastructure that was originally developed for Java stream pipelines.

The results show that the proposed approach can successfully capture trace data suitable for visualizing LINQ query execution and can make intermediate results, element transformations, and final results easier to understand. In this way, the system can support both learners and educators by providing a concrete visual representation of LINQ execution. At the same time, the current implementation still has limitations: it is restricted to LINQ to Objects, and JavaWiz integration currently requires manual JSON import. Run-time overhead is acceptable for educational examples, but trace persistence noticeably increases execution time.

In conclusion, the thesis provides a solid basis for the educational visualization of LINQ queries by combining IL-level instrumentation, run-time trace generation, and visualization. It also lays a foundation for future extensions of both the tracing system and the JavaWiz integration.

Appendices

A Non-simplified IL Example

```
1 stloc.0
2
3 ldloc.0
4 ldsfld      class [System.Runtime]System.Func`2<int32, bool>
5             BScThesisExamples.Program/'<>c'::'<>9__0_0'
6 dup
7 brtrue.s   IL_0061
8 pop
9
10 ldsfld      class BScThesisExamples.Program/'<>c' BScThesisExamples.Program/'<>c'::'<>9'
11 ldftn      instance bool BScThesisExamples.Program/'<>c'::'<Main>b__0_0'(int32)
12 newobj     instance void class
13             [System.Runtime]System.Func`2<int32, bool>::ctor(object, native int)
14 dup
15
16 stsfld     class [System.Runtime]System.Func`2<int32, bool>
17             BScThesisExamples.Program/'<>c'::'<>9__0_0'
18 IL_0061: nop
19 stloc.s    predicate
20
21 ldstr      "Where"
22 ldc.i4.0
23 ldc.i4.0
24 ldstr      "x => x < 5"
25 call       class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>
26             TraceLinq.TraceLinqExtensions::TraceLinq<int32>
27             (class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>,
28              string, valuetype TraceLinq.TraceLinqExtensions/TracePhase, bool, string)
29
30 ldloc.s    predicate
31 call       class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>
32             [System.Linq]System.Linq.Enumerable::Where<int32>
33             (class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>,
34              class [System.Runtime]System.Func`2<!!0/*int32*/, bool>)
35
36 ldstr      "Where"
37 ldc.i4.1
38 ldc.i4.0
39 ldstr      "x => x < 5"
40 call       class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>
41             TraceLinq.TraceLinqExtensions::TraceLinq<int32>
42             (class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>,
43              string, valuetype TraceLinq.TraceLinqExtensions/TracePhase, bool, string)
44
45 ldsfld     class [System.Runtime]System.Func`2<int32, int32>
46             BScThesisExamples.Program/'<>c'::'<>9__0_1'
47 dup
48 brtrue.s   IL_00ac
49 pop
50
51 ldsfld     class BScThesisExamples.Program/'<>c' BScThesisExamples.Program/'<>c'::'<>9'
52 ldftn     instance int32 BScThesisExamples.Program/'<>c'::'<Main>b__0_1'(int32)
53 newobj     instance void class
54             [System.Runtime]System.Func`2<int32, int32>::ctor(object, native int)
55 dup
56
57 stsfld     class [System.Runtime]System.Func`2<int32, int32>
58             BScThesisExamples.Program/'<>c'::'<>9__0_1'
59 IL_00ac: nop
```

```

60 stloc.s    selector
61
62 ldstr      "Select"
63 ldc.i4.0
64 ldc.i4.0
65 ldstr      "x => x * 2"
66 call       class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>
67           TraceLinq.TraceLinqExtensions::TraceLinq<int32>
68           (class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>,
69           string, valuetype TraceLinq.TraceLinqExtensions/TracePhase, bool, string)
70
71 ldloc.s    selector
72
73 call       class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!1/*int32*/>
74           [System.Linq]System.Linq.Enumerable::Select<int32, int32>
75           (class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>,
76           class [System.Runtime]System.Func`2<!!0/*int32*/, !!1/*int32*/>)
77
78 ldstr      "Select"
79 ldc.i4.1
80 ldc.i4.0
81 ldstr      "x => x * 2"
82 call       class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>
83           TraceLinq.TraceLinqExtensions::TraceLinq<int32>
84           (class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>,
85           string, valuetype TraceLinq.TraceLinqExtensions/TracePhase, bool, string)
86
87 call       class [System.Collections]System.Collections.Generic.List`1<!!0/*int32*/>
88           TraceLinq.TraceLinqExtensions::ToListTrace<int32>
89           (class [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>)

```

Listing 18: Non-simplified instrumented representation of the running example shown in Listing 10.

List of Tables

1	Execution times and overhead of the base LINQ query.	33
2	Execution times and overhead of the LINQ query with SelectMany	33
3	Sizes of the generated JSON trace files for the two evaluated LINQ queries. . . .	34

List of Figures

1	Preview of a LINQ query execution visualized in JavaWiz.	6
2	System architecture and data flow of the LINQ visualization system, from the compiled C# program through instrumentation and run-time tracing to the final visualization in JavaWiz.	7
3	Object-oriented representation of LINQ operations in the trace model.	10
4	Conceptual structure of the internal trace model.	11
5	Instrumentation log for the running example, showing the merge of the trace subsystem, the detection of LINQ operations, and the recognition of unsupported Parallel LINQ calls.	22
6	Visualization pipeline that transforms a recorded JSON trace into an internal LINQ trace representation and finally into the JavaWiz visualization structures. .	23
7	Final JavaWiz visualization of the running example, showing that elements 1 and 3 pass the filter, are transformed to 2 and 6, and are collected into the final list. .	27
8	Workflow diagram for visualizing a LINQ query.	28
9	Console output of the instrumentation step, showing that the LINQ operations Where , Select , and ToList were detected and instrumented with tracing functionality.	29
10	JavaWiz import dialog for submitting the generated JSON trace.	29
11	JavaWiz visualization of the running example at step six, showing that element 1 passes the filter while element 8 is removed by the Where operation.	30
12	Final JavaWiz visualization of the running example, showing the complete query pipeline, the intermediate results, and the final output [2, 6].	30
13	JavaWiz visualization of the flattening example at step 7, showing how the first nested list is flattened and counted without removing any duplicates.	31
14	Final JavaWiz visualization of the flattening example, showing that the duplicate value 2 is removed by Distinct and the final count is 3.	31

Listings

1	Simple example of LINQ method syntax.	2
2	Simple example of LINQ query syntax.	3
3	LINQ query used as the running example for trace generation and visualization. .	14
4	JSON output of the recorded LINQ query trace.	15
5	ILRepack invocation for merging the trace subsystem into the target DLL.	18
6	Simplified example of instrumentation for an intermediate LINQ Where operation. .	19
7	Simplified example of instrumentation for a terminal LINQ ToList operation. . .	20

8	Running example LINQ query used to illustrate the instrumentation process. . .	20
9	The simplified compiled representation of the example query.	20
10	Simplified instrumented representation of the example query.	21
11	Conversion function for transforming trace data into visualization information. .	24
12	Running example LINQ query used to illustrate the visualization process in JavaWiz.	25
13	Simplified imported StreamOperationValue map excerpt.	25
14	Simplified visualization data excerpt.	26
15	LINQ query that filters and transforms a collection.	29
16	LINQ query that flattens collections and then deduplicates the resulting collection.	30
17	Evaluated LINQ queries with alternative terminal operations.	32
18	Non-simplified instrumented representation of the running example shown in Listing 10.	39

References

- [1] S. M. Cisar, R. Pinter, and D. Radosav. Effectiveness of program visualization in learning java: a case study with jeliot 3. *Int. J. Comput. Commun. Control*, 6(4):668–680, 2011. DOI: <https://www.doi.org/10.15837/IJCCC.2011.4.2094>.
- [2] S. Cropp. Fody/Home, Apr. 2026. URL: <https://github.com/Fody/Home>.
- [3] J. Evain. Mono.Cecil | Mono. URL: <https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/>.
- [4] gewarren. What is managed code? - .NET. URL: <https://learn.microsoft.com/en-us/dotnet/standard/managed-code>.
- [5] J. Katona, A. Kovari, I. Heldal, C. A. Costescu, A. Rosan, R. Demeter, S. Thill, and T. Stefanut. Using eye-tracking to examine query syntax and method syntax comprehension in LINQ. In *11th IEEE International Conference on Cognitive Infocommunications, CogInfoCom 2020, Mariehamn, Finland, September 23-25, 2020*, pages 437–444. IEEE, 2020. DOI: <https://www.doi.org/10.1109/COGINFocom50765.2020.9237910>.
- [6] M. Koch. Interceptors – Using C# 12 in Rider and ReSharper | The .NET Tools Blog, Jan. 2024. URL: <https://blog.jetbrains.com/dotnet/2024/01/16/interceptors-using-csharp-12-in-rider-and-resharper/>.
- [7] E. Lahtinen, K. Ala-Mutka, and H. Järvinen. A study of the difficulties of novice programmers. In J. C. Cunha, W. M. Fleischman, V. K. Proulx, and J. Lourenço, editors, *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2005, Caparica, Portugal, June 27-29, 2005*, pages 14–18. ACM, 2005. DOI: <https://www.doi.org/10.1145/1067445.1067453>.
- [8] Microsoft. Enumerable.Count Method (System.Linq). URL: <https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.count?view=net-10.0>.
- [9] Microsoft. Enumerable.Distinct Method (System.Linq). URL: <https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinct?view=net-10.0>.
- [10] Microsoft. Enumerable.Select Method (System.Linq). URL: <https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select?view=net-10.0>.
- [11] Microsoft. Enumerable.SelectMany Method (System.Linq). URL: <https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany?view=net-10.0>.
- [12] Microsoft. Enumerable.ToList<TSource>(IEnumerable<TSource>) Method (System.Linq). URL: <https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.tolist?view=net-10.0>.
- [13] Microsoft. Enumerable.Where Method (System.Linq). URL: <https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where?view=net-10.0>.

- [14] Microsoft. Introduction to LINQ Queries - C#. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/linq/get-started/introduction-to-linq-queries>.
- [15] Microsoft. IQueryable Interface (System.Linq). URL: <https://learn.microsoft.com/en-us/dotnet/api/system.linq.iqueryable?view=net-10.0>.
- [16] Microsoft. Language Integrated Query (LINQ) - C#. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/linq/>.
- [17] Microsoft. runtime/src/libraries/System.Linq/src/System.Linq at main · dotnet/runtime. URL: <https://github.com/dotnet/runtime/tree/main/src/libraries/System.Linq/src/System.Linq>.
- [18] Microsoft. What's new in C# 12. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-12>.
- [19] Microsoft. Write LINQ queries - C#. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/linq/get-started/write-linq-queries>.
- [20] M. Weninger, S. Grünbacher, and H. Prähofer. Javawiz: A trace-based graphical debugger for software development education. In *33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*, pages 1–12. IEEE, 2025. DOI: <https://www.doi.org/10.1109/ICPC66645.2025.00023>.
- [21] R. Wojszczyk, A. O. Hapka, and T. Królikowski. Performance analysis of extracting object structure from source code. In G. A. Tsihrintzis, C. Toro, S. A. Ríos, R. J. Howlett, and L. C. Jain, editors, *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 27th International Conference KES-2023, Athens, Greece, 6-8 September 2023*, volume 225 of *Procedia Computer Science*, pages 4065–4073. Elsevier, 2023. DOI: <https://www.doi.org/10.1016/J.PROCS.2023.10.402>.