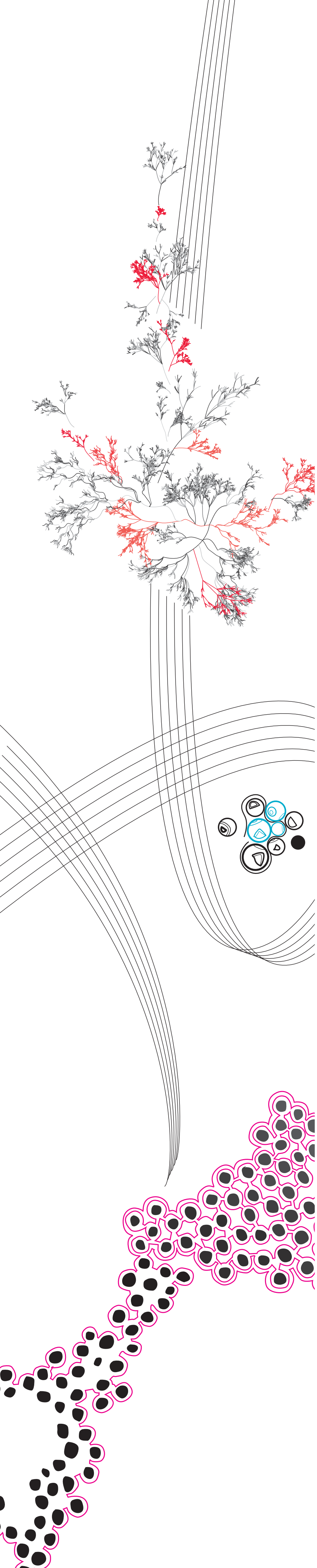


MASTER THESIS

The LAMP Framework
A language-agnostic code quality assurance framework for multi-paradigm languages

Author:
Marnick Q.T.P. van der Arend

Master of Computer Science
Faculty of Electrical Engineering, Mathematics & Computer Science (EEMCS)
Software Technology specialisation

Supervisors:
dr.ir. Vadim Zaytsev
Jan-Jelle Kester, MSc. (Info Support B.V.)
Rinse van Hees, MSc. (Info Support B.V.)

March 13, 2023

UNIVERSITY OF TWENTE.

Abstract

We can assess the quality of code to reduce the probability that software leads to unwanted behaviour and faults. In recent years, object-oriented programming (OOP) languages have adopted many features of the functional programming (FP) paradigm. We call the combination of these programming paradigms multi-paradigm (MP). Many tools have been built that measure code quality specifically for one MP programming language, but a large heterogeneous data set capturing important paradigm concepts, or the combination thereof, was lacking for each language. We hypothesise that we can create a language-agnostic representation of MP programming languages that can capture paradigm concepts and constructs. As a result, the language-agnostic representation can be used to reach a larger potential data set of usable projects.

To find the correct properties to create a language-agnostic representation of MP programming languages, we analyse paradigm constructs present in Java, C#, Kotlin and Scala. Using the Goal Question Metric approach, we create a mapping between code quality characteristics and code quality metrics. Using the analysis of selected MP programming languages and a set of code quality metrics, we design a language-agnostic representation of MP programming languages, namely, the LAMP metamodel. Using the metamodel, we created a framework for transforming MP programming languages to the LAMP metamodel and computing metrics on this language-agnostic representation. To evaluate the correctness and completeness of the LAMP metamodel, we evaluate how each metric can be computed using the metamodel representation. We have developed a prototype of our framework that can transform a Java project's source code into a language-agnostic representation. To evaluate the accuracy of our framework prototype, we compared metric computations of our prototype with two benchmarks of five Java projects.

From our evaluation, we conclude that our LAMP metamodel is capable of capturing the most important constructs of MP programming languages, and our framework is able to put forward a consistent workflow to assure code quality. Therefore, we argue that our framework design represents a significant step towards solving the data scarcity problem with fault proneness detection in MP programming languages.

Acknowledgements

To begin, I'd like to express my gratitude to my supervisors, Jan-Jelle Kester (Info Support), Rinse van Hees (Info Support), and Vadim Zaytsev (University of Twente). The writing of this thesis has been greatly assisted by their questions, feedback, and insights.

Moreover, I would like to express my gratitude to Info Support for supplying the introductory points and hosting this thesis. I would like to thank Bram Janssens (Info Support) for an introductory course into Kotlin, which aided developing the prototype.

Finally, I would like to express my appreciation to my family and friends for all the support they have provided, as well as the fact that they were always willing to listen.

- Marnick

Contents

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CI/CD	Continuous Integration and Continuous Delivery
CST	Concrete Syntax Tree
FP	Functional Programming
GPL	General-Purpose Programming Language
GQM	Goal Question Metric
IDE	Integrated Development Environment
JVM	Java Virtual Machine
LINQ	Language-Integrated Query
MM	Maintainability Model
MP	Multi-Paradigm
OOP	Object-Oriented Programming
SDK	Software Development Kit
SIG	Software Improvement Group
SQA	Software Quality Assurance
XML	Extensible Markup Language
XSD	XML Schema Definition

1 Introduction

Programming languages come in many shapes and sizes. They are categorised into one or more programming paradigms. Every programming paradigm has the purpose of solving a certain problem with the optimal solution [1]. A large number of popular languages have adopted multiple programming paradigms within their language. While the combination of multiple paradigms can solve a problem in several ways, it can also produce new combinations of constructs that were not intended to be combined. This can lead to unwanted behaviour of the system.

In recent years, object-oriented programming languages (e.g. Java, C#, Kotlin, Scala) have added features inspired by the functional programming paradigm. These languages rank high on the popularity index [2] and are widely used for enterprise software development. In this thesis, we call the combination of these two programming paradigms multi-paradigm (MP).

To reduce the probability of unwanted behaviour in software systems, its quality can be assessed. Low-quality software can lead to more maintenance than high-quality software [3]. Consequently, when quality is left unaddressed for too long, evolving systems can decrease in quality to a point where it is better to rebuild the whole system. To avoid costly remediation and potentially catastrophic events [4], it is most important to have a notion of software quality before it reaches production. It is the job of a good software engineer to prevent this from happening and to maintain software quality at an acceptable standard [5].

There are many methods that can be used to measure software quality, such as analysing the development process, the quality of a system in use, or the quality of code [6]. When measuring software quality from a maintenance point of view, code quality can give a good indication of the amount of maintenance required. Code quality assessment can be done as an automatic process with static code analysis (during compile-time) or dynamic code analysis (during run-time).

Measurement of code quality in object-oriented or functional programming languages is no new feat [7, 8, 9, 10]. However, the impact of the combination of both paradigms on software quality has not received much attention in the academic literature [11, 12, 13]. Moreover, when object-oriented and functional styles melt together, they might introduce new kinds of complexity which impacts code quality. Due to the lack of literature in this area, developers may not be aware of new ways in which software faults can be introduced. For that reason, it is important to focus on multi-paradigm software quality.

In previous work, the fault proneness (i.e. unwanted behaviour) was predicted for projects in a particular MP programming language, namely Scala and C#. Code metrics describing important language constructs were fed into a prediction model along with bug issue tracker data. Due to the lack of data on multi-paradigm constructs within mature projects written in each of these languages, the precision of these prediction models was poor [11, 12, 13]. For accurate predictions of fault proneness, a larger heterogeneous data set is required that captures all paradigm concepts.

To solve the issue of data scarcity, we argue that multi-paradigm programming languages share many similar features. Therefore, we hypothesise that we can create a language-agnostic representation of multi-paradigm programming languages. As a result, the language-agnostic representation can be used to reach a larger potential data set of usable projects while still being able to measure the code quality for individual languages.

For that reason, this research explores the design of a language-agnostic code quality assurance framework that can assess the quality of source code for MP languages. Using this representation as a basis of the framework, several workflows are defined to transform programming language constructs into the language-agnostic representation and to extract code metrics that describe important paradigm constructs. We hypothesise that these metrics can be computed as accurately using a language-agnostic representation as with language-specific code quality assessment tools.

1.1 Research Questions & Approach

To validate our hypothesis and structure this thesis, we present our main research question and its subquestions below. Every subquestion contains an approach to answering that subquestion. The main research question and its subquestions are summarised and answered in Section 8.1.

RQ: To what extent can a language-agnostic code quality assurance framework be designed for multi-paradigm languages?

RQ1 *What are the constructs of multi-paradigm programming languages?*

To correctly create a language-agnostic representation of multi-paradigm programming languages, it is important to know what the most important constructs of MP languages are. We analyse the MP programming languages Java, C#, Kotlin, and Scala due to their wide use in enterprise software development and popularity. These languages implement features of OOP and FP in their own way; therefore, it is useful to compare these languages and discover similarities and differences. With this information, we can make informed decisions about the abstraction from language-specific constructs to a language-agnostic representation.

RQ2 *Which code quality characteristics must the framework capture, and how?*

Before we can create a language-agnostic representation of the MP programming languages, it is important to know how code quality should be measured. Using the Goal Question Metric approach, we create a mapping between code quality characteristics and metrics that can provide insights to those characteristics. With the measurement requirements of these metrics, we can distil the information that is required from each language construct. As a result, we can make informed decisions about what constructs must be modelled in the language-agnostic representation.

RQ3 *How can we measure code quality for multi-paradigm languages in a language-agnostic manner?*

With all prerequisites gathered from RQ1 and RQ2, the language-agnostic representation of MP programming languages can be designed. Using the representation, a workflow is designed that can followed to transform source code from a MP language to the language-agnostic representation to measure code quality on. In that design, we incorporate instructions on how to measure code metrics with the language-agnostic representation. These designs show how code quality can be measured in a language-agnostic manner.

RQ4 *How does this framework perform in comparison to language-specific analyses?*

To evaluate our framework, a prototype implementation is developed that complies with the design of the framework. The evaluation includes three evaluation criteria to evaluate framework performance, namely metric evaluation, framework evaluation, and benchmark evaluation. To evaluate the completeness of the computation of selected metrics, we cover for each metric how their requirements map onto the language-agnostic representation. To evaluate the framework, we cover the complexities of taking a language-agnostic approach when measuring code quality. To evaluate the correctness of our framework, the computed language-agnostic metric results are compared to language-specific metric results from benchmark code analysis tools. Using the comparison, we can validate if our language-agnostic representation captures all required constructs to measure code quality on a language-agnostic level.

1.2 Host Organisation

This thesis research was carried out at the Info Support B.V. Research Centre in Veenendaal, The Netherlands. Info Support is an IT consulting firm based in The Netherlands and Belgium with 500+ consultants. They operate in the sectors of finance, industry, mobility & public, healthcare, agriculture and managed services. Info Support has a research centre that researches topics relating to AI, software architecture, and software development methodologies.

1.3 Contributions

Before and during the execution of this project, several contributions were made to aid the academic body of knowledge.

- In the preparatory phase of this thesis, the paper describing the initial approach for researching this topic was written for BENEVOL 2022 [14]. The work was presented during the event on 13 September 2022 in Mons, Belgium.
- Described an extensive feature analysis of major multi-paradigm programming languages (Appendix A).
- Detailed a comprehensive cross-comparison of Java, C#, Kotlin, and Scala (Section 2.2).
- Provided background information for this thesis on programming paradigms, abstract representations of programming languages, metaprogramming, software quality, and code quality metrics & smells (Section 2).
- Defined a mapping between the goal of measuring multi-paradigm code quality, its quality characteristics, and associated metrics (Section 3).
- Designed a language-agnostic representation of multi-paradigm programming languages (Section 4).
- Developed the open-source LAMP Framework prototype[1] written in Kotlin as described in Section 5. Currently, it supports transformations to the language-agnostic

[1] `https://github.com/MarnickvdA/LAMP-Framework`

representation for Java projects. It uses the LAMP metamodel XSD representation listed in Appendix C.

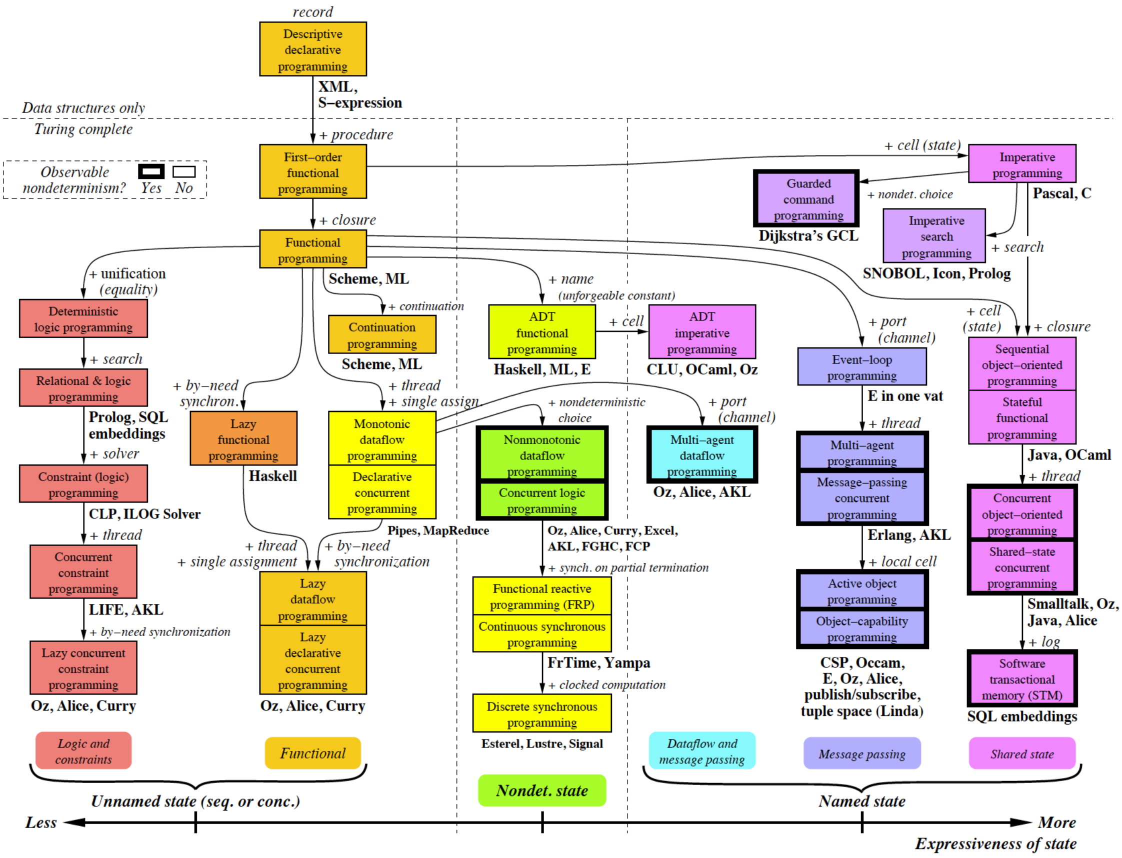
1.4 Outline

This thesis is structured as follows. In Section 2, we provide a background on programming paradigms, software quality, language compilation, and code metrics & smells. Section 3 describes the mapping of our code quality assurance goal to quality characteristics and metrics. Section 4 presents the language-agnostic representation of multi-paradigm programming languages. Section 5 presents the design of our framework workflow and a description of how to compute metrics with the language-agnostic representation. Section 6 defines our evaluation method, the evaluation of metric computations for several Java projects, and the evaluation of limitations of the framework. Section 7 discusses related work from which inspiration was drawn for this thesis. Finally, we conclude the thesis in Section 8 by presenting our findings, discussing design choices, and describing future work.

2 Background

This section covers the subjects that we deem necessary for understanding the intricacies of a language-agnostic multi-paradigm code quality assurance framework. In Section 2.1, we explain the object-oriented and functional paradigms. It is followed by a cross-comparison of four multi-paradigm languages in Section 2.2. In Section 2.3, the characteristics of software quality, code quality, and its measurement are described. In Section 2.4, we describe the language compilation process, which aids understanding the transformation of MP programming languages to the language-agnostic representation. Finally, a description of well-known code metrics and smells is provided in Section 2.5.

2.1 Programming Paradigms



record
Descriptive declarative programming
XML, S–expression
Data structures only
Turing complete
+ procedure
Observable nondeterminism? Yes No
First–order functional programming
+ cell (state)
Imperative programming
Guarded command programming
+ nondet. choice
Pascal, C
Dijkstra's GCL
+ closure
Functional programming
Imperative search programming
+ search
SNOBOL, Icon, Prolog
+ unification (equality)
Scheme, ML
+ name (unforgeable constant)
+ continuation
Deterministic logic programming
Continuation programming
ADT functional programming
+ cell
ADT imperative programming
+ port (channel)
+ cell (state)
+ closure
Scheme, ML
Haskell, ML, E
CLU, OCaml, Oz
Event–loop programming
Sequential object–oriented programming
+ search
Relational & logic programming
+ by–need synchron.
+ thread
+ single assign.
+ nondeterministic choice
+ port (channel)
E in one vat
Stateful functional programming
Prolog, SQL embeddings
Lazy functional programming
Monotonic dataflow programming
Nonmonotonic dataflow programming
Multi–agent dataflow programming
+ thread
Java, OCaml
+ solver
Haskell
Declarative concurrent programming
Concurrent logic programming
Oz, Alice, AKL
Multi–agent programming
Message–passing concurrent programming
+ thread
Constraint (logic) programming
Concurrent object–oriented programming
CLP, ILOG Solver
Pipes, MapReduce
Oz, Alice, Curry, Excel, AKL, FGHC, FCP
Erlang, AKL
Shared–state concurrent programming
+ thread
+ thread
+ single assignment
+ by–need synchronization
+ synch. on partial termination
+ local cell
Concurrent constraint programming
Lazy dataflow programming
Functional reactive programming (FRP)
Active object programming
Smalltalk, Oz, Java, Alice
LIFE, AKL
Continuous synchronous programming
Object–capability programming
+ by–need synchronization
Lazy declarative concurrent programming
FrTime, Yampa
+ log
Lazy concurrent constraint programming
+ clocked computation
CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)
Software transactional memory (STM)
Oz, Alice, Curry
Oz, Alice, Curry
Discrete synchronous programming
SQL embeddings
Esterel, Lustre, Signal
Logic and constraints
Functional
Dataflow and message passing
Message passing
Shared state
Nondet. state
Unnamed state (seq. or conc.)
Named state
Less
More
Expressiveness of state

Figure 1: Taxonomy of Programming Paradigms [1]

There are many programming paradigms in existence, as shown in Figure 1. These programming paradigms are often influenced by other paradigms, or built as an extension of another. Each programming paradigm has a place in the world of paradigms, as they can solve a problem with a fitting solution. It can be the simplest, easiest to reason about, or the optimal solution [1]. As a software program grows in size, it is often the case that different types of problems are solved. Therefore, a programming language supports multiple programming paradigms to allow a developer to solve these problems with the most appropriate solutions. These languages are generally known to be multi-paradigm languages, more commonly known as general purpose languages (GPLs).

As we have learnt in the Introduction, multi-paradigm within this thesis covers the object-oriented paradigm and the functional paradigm. The object-oriented paradigm is introduced in the taxonomy on the rightmost side of expressiveness of state (x-axis). The functional paradigm is positioned on the far left side of the x-axis.

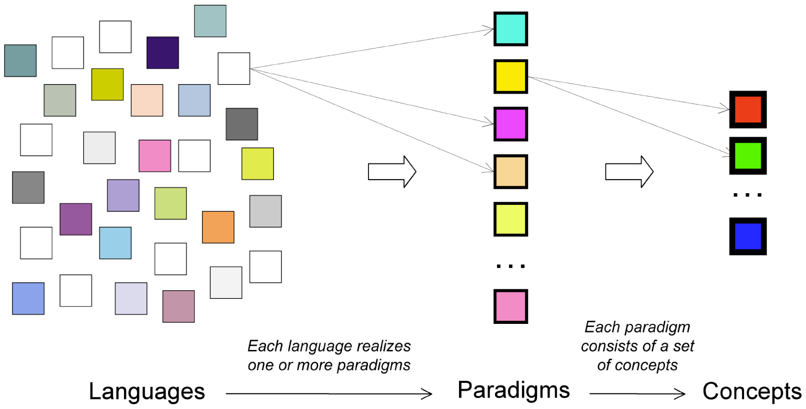

Each language realizes one or more paradigms
Each paradigm consists of a set of concepts
Languages
Paradigms
Concepts

Figure 2: Languages, Paradigms, and Concepts [1]

As shown in Figure 2, languages realise one or more paradigms, and every programming paradigm consists of a set of concepts. Due to our language-agnostic approach to measuring code quality, we should know the concepts that embody the paradigms that are part of the multi-paradigm setting. When these paradigm-specific concepts are identified, we can generalise language-specific implementations that lead to a language-agnostic representation.

As depicted in Figure 1, it is clear that paradigms flow into each other and take additional concepts into account (e.g., procedures and closures). There are three concepts that are interesting to analyse with our definition of multi-paradigm.

We begin at the top of this taxonomy, where *Record* is found. A record is the most basic part of a paradigm. It is a data structure. It is a group of references to data items with indexed access to each item. Many of the data structures that developers use every day (e.g., arrays, lists, trees, hash tables) are derived from records [1].

We proceed to *Closure* that is part of the functional and object-oriented programming paradigm. A closure is short for a lexically scoped closure. From a developer point of view, it can be seen as a "packet of work". A closure can combine a procedure with external references such as records to form a powerful tool for programming. A closure has two sets of references: a closed environment (from the definition) and the arguments (for each call) [1]. A closure thus takes some input, does operations with that input, and outputs a result, i.e. the return value.

Lastly, the taxonomy covers *Named State* for imperative programming languages. Named state can be referred to as the abstract notion of time. It is philosophised that a property that changes over time should be able to keep the same property name, while its contents might change over the course of the program. Named state can help to improve the modularity of a program, where a part of a system can be changed without changing the entire system [1].

In the following sections, we will explain the concepts of object-oriented programming and functional programming to give insight into the way the languages supporting them are structured. Once the general concepts are covered, we can observe the similarities and differences between paradigms and the languages that realise them more easily.

2.1.1 Object-Oriented Programming (OOP)

The OOP paradigm is an extension of the imperative programming paradigm. The object-oriented programming term was originally coined by Dr. Alan Kay in 1966 describing an architecture for messaging where objects pass on messages using procedures (i.e. methods) [15]. In Figure 1, the OOP paradigm is on the most right side of expressiveness of state with the combination of named state and closures. OOP is defined in the book Object-Oriented Analysis and Design with Applications as:

"a method of implementation in which programs are organised as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships" [16]

Real-world phenomena within programs can be modelled using classes. As explained in Section 2.1, closures can be used to write packets of work (i.e., functions) that return some value. In the case of OOP, you could see the function as a *class constructor* and the returned value is called the *object* or *class instance.*

- *Classes & Objects*: at the core of OOP are the classes and its associated objects. Classes act as blueprints for their associated objects, describing fields and methods. Objects are instances of classes containing defined data [17]. Classes can also contain static fields and methods that provide singleton behaviour across multiple instances.

- *Encapsulation*: OOP uses classes to encapsulate data and the methods that operate on those data. It is used to protect the private information of that class and only expose functionality that is made public [16]. Most OOP languages use access modifiers to indicate the accessibility of encapsulated data.

- *Inheritance*: a class within OOP can inherit (much of) the functionality from another class when it "extends" that class. This hierarchy can exist on multiple levels where a class has one or more children [16]. The most common types of inheritance are single inheritance, multi-level inheritance and hierarchical inheritance.

- *Polymorphism*: it translates to "having multiple forms" from Greek. In OOP, it means that one class at the root of the hierarchy lays the blueprint for a set of classes with similar behaviour [16]. E.g., the `Bus` class and `Car` class both inherit from `Vehicle` and this *parent* class can contain the `drive()` method which applies to both `Bus` and `Car` (i.e. subtype polymorphism [18]). It allows for generic type declaration on a class, interface, or method where it is unknown which precise type will be used. Parametric polymorphism, commonly known as overloading, can be used to create one function that can interact with multiple types. Another common type of polymorphism is casting, which forces the transformation of one type to another. The final type of polymorphism is ad hoc polymorphism that allows functions with the same name to act differently for different types. A common addition to the types of polymorphism is the appliance of variance to the generic types. Variance is used to project bounds onto generic types. There are four categories of variance. Invariance where no supertypes or subtypes are accepted, covariance where no supertypes are accepted but subtypes are, contravariance where supertypes are accepted but subtypes are not, and bivariance where supertypes and subtypes are accepted [19].

2.1.2 Functional Programming (FP)

The FP paradigm is at the core of the taxonomy of programming paradigms. It originates from Lambda Calculus [20], a formal mathematical logic system that enables one to express computation based on function abstraction and application using variable binding and substitution [21]. At the core of functional programming lies immutability, on which other features of FP depend. There are many benefits that FP brings to the table, which are covered in the following bullet points.

- *Functions as First-Class citizens*: functions are seen as first-class citizens, which means they can be used as values within the source code. Functions can be assigned to parameters, passed as return values, or stored in a data structure. In FP, the entire program is a function that can contain other functions to make the program more modular, but it finally produces a result [21]. Therefore, we can nest functions within other functions. This is a common tool for dividing a function into readable chunks and increase reusability of source code.

- *Lambda functions*: a lambda function is sometimes referred to as an anonymous function, as it is a function without a name. Lambda functions are functions that are based on Lambda Calculus [20]. Functions can be seen as a closure since lambda functions take some input, perform some mathematical computation, and output a value [1]. Since functions can be provided as output, we can create functions that contain functions as parameter values or return a function. This type of function is called a higher-order function. Hence, the internal behaviour of a (higher-order) function can change depending on one or more externally described functions as parameter value. This allows for reusability of often used functions [12].

- *Referential transparency*: functions in FP provide a mapping from an input to an output. It will always output the same value when the same input is provided. Therefore, when you refer back to an expression in a later part of the program, it is a certainty that the output is the same as before. You could change the expression with its value or vice versa and it would not affect the behaviour of the program [21]. This eliminates any occurrence of side effects [22].

- *Recursion*: looping over collections in FP is performed via recursion instead of the imperative approach of sequencing. In recursive functions, the named state is explicitly passed through via the argument(s) of the function until the base case is reached. Thereafter, this state is passed as output to the function caller until the start of the recursive operation is reached, thereby completing the chain [23]. Recursion is used for well-known operations on collections such as `map`, `reduce` and `filter`. To avoid the allocation of a new stack frame for a function, we can employ *tail-call optimisation* to a recursive function that makes the function memory safe and performant [24].

- *Lazy evaluation*: non-strict semantics or lazy evaluation, often also called *call by need* has a key feature that it only evaluates the function when it is called. Lazy evaluation allows the developer to be more expressive and relieves them of concerns about evaluation order. It allows you to postpone evaluation of potentially resource-intensive tasks, like querying unbounded data structures [23].

- *Pattern matching*: it is a form of syntactic sugar within the FP paradigm that allows a developer to write multiple equations within a function. In this type of function, only one of the equations is applicable in a given situation [23].

- *Currying*: currying can be done by splitting a function that takes multiple arguments into separate functions with one argument. With a FP function as first-class citizen, functions can be returned as results in another function. Therefore, function calls can be chained. Therefore, the transformation of creating single-argument functions from a multi-argument function is called currying [25], named after the mathematician Haskell Curry. Currying is a type of partial application. Partial application transforms a multi-argument function into a smaller-arity function.

2.2 Cross-Comparison of MP Languages

To limit the scope of this thesis, we selected only a few programming languages that are regarded MP. We chose statically-typed MP programming languages that are widely used in the industry and support functional programming concepts to different extends. To give the best representation of the current state of multi-paradigm constructs, we chose to target recent versions of every language. We have selected the following versions of the selected languages with their respective release dates sorted by popularity [2]:

- Java 17, released in September 2021
- C# 10, released in July 2022
- Kotlin 1.7.0, released in June 2022
- Scala 3, released in May 2021

We analysed how the selected languages adopted the concepts of OOP and FP. This allows us to see patterns within the implementation for each selected language. However, this information is too broad for the general understanding of this thesis. Therefore, based on the level of interest, the feature analysis per language can be read in Appendix A.

In the following, we will describe the cross-comparison of constructs in our languages to see how they are similar or different. It is essential to know these details to be able to correctly create an abstraction from these specific languages.

2.2.1 Encapsulation

All languages contain the access modifiers `public`, `protected` and `private`. C# and Kotlin also contain an `internal` access modifier, which focuses on the encapsulation with respect to assemblies or modules (i.e., compilation units). While Java supports modules, they do not have an access modifier for them because the encapsulation is defined at the module declaration. Scala and Kotlin provide the least options to restrict accessibility. Java and C# provide more detailed and specific scenarios with combinations of several access modifiers (e.g., `protected internal`, `protected private`).

The definition of each access modifier differs between programming languages. There are some special access modifiers that do not exist in most languages. Every language provides full access with the `public` access modifier. The `protected` access modifier behaves the same for Java, C# and Kotlin, but in Scala it does not allow to access a class from a subclass in another package. The `private` access modifier means the same in Java, C# and Scala, but in Kotlin `private` means file-private, and not class-private. When no access modifier is explicitly provided, the compiler-assigned access modifier is different for some languages. Java limits access to the package, while C# limits access to the assembly, for Scala and Kotlin the default access modifier is `public`.

2.2.2 Classes

On a generic level, every language contains a base set of functionality for its classes, called class members. We have summarised the available class members for each language in Table 1.

Table 1: Class member overview

	Java	C#	Kotlin	Scala
Constructor	Yes	Yes	Yes	Yes
Constructor in class header	No	No	Yes	Yes
Secondary constructor(s)	Yes	Yes	Yes[1]	Yes[1]
Instance initializer	Yes	No	Yes	No
Static initializer	Yes	Yes[1]	No	No
Companion object	No	No	Yes	Yes[1]
Field(s)	Yes	Yes	Yes	Yes
Property(s)	No	Yes	Yes	No
Method(s)	Yes	Yes	Yes	Yes
Nested class(es)	Yes[1]	No	Yes	No
Inner class(es)	Yes[1]	Yes	Yes[1]	Yes
Interface(s)	Yes	Yes	No	No
Constant(s)	Yes	Yes	Yes	Yes
Object(s)	No	No	Yes	Yes
Operator(s)	No	Yes	Yes	Yes

[1] Additional information described below.

There are several members in Table 1 that behave differently in contrast to the same member in other languages. These cases are listed below.

- Secondary constructors in Kotlin and Scala must include a call to the primary constructor.
- The static initializer in C# is a static constructor.
- The companion object in Scala must be declared in the same file as the companion's class, but cannot reside inside the class declaration.
- There is a difference between a nested class and an inner class. A nested class is not bound to an outer class instance but is declared inside the outer class. An inner class is bound to the outer class instance. In Java, a nested class is declared inside the outer class as a static class. In Kotlin, the `inner` keyword is reserved for an inner class. As a result, non-inner classes are static by default.

There is a difference in the way class members are defined between Scala and on the other side Java, C#, and Kotlin. While in Scala everything is an expression, the others have a specific set of class members. Scala can be customised more, allowing constructs to be expressed in more ways than in other languages. Therefore, Scala might make it more complex to express it in a language-agnostic metamodel.

2.2.3 Objects

Objects show the same behaviour in the selected languages. They are the instance of a class, although this is not materialised at the syntax level for Java and C#. Kotlin and Scala have a special `object` keyword to declare objects that follow the Singleton pattern [26]. Another type of object in Kotlin and Scala is the companion object, which is tied to one class. It is a container that describes static properties and methods for a class.

This functionality is similar to static members in Java and C#, but it is encapsulated syntactically.

2.2.4 Inheritance

Every selected language supports single inheritance, multi-level inheritance, and hierarchical inheritance between classes. Class composition is made possible with Interfaces in Java, C# and Kotlin, or with Traits in Scala.

Java, C#, and Scala allow classes to be inherited by default. In Kotlin, every class is final by default which restricts inheritance, unless the `open` modifier is used to allow inheritance. For Java, C#, and Scala, the inheritance can be restricted with the `sealed` modifier. Java also allows the restriction of inheritance with the `final` modifier, and only with the `sealed` and `permits` modifiers one can specify which classes are allowed to inherit from the sealed class.

2.2.5 Polymorphism

For polymorphism, we focus on method overriding, method overloading, operator overloading, type parameterisation, and variance.

All languages are also able to restrict which methods can be overridden, or signal that a particular method must be overridden. By default, C# and Kotlin do not allow overrides. To restrict the overriding of a method, we must apply the `final` modifier in Java and Kotlin, the `sealed` modifier in C#, and the `final` modifier in Kotlin only if the super method was marked `open`. To indicate that a method must be overridden in a non-abstract subclass, Java, C#, Scala, and Kotlin use the `abstract` modifier. To signal that a method can be overridden, in C# the `virtual` modifier is used, and in Kotlin the `open` modifier is used.

For all languages, method overloading is possible by changing the number of parameters, the data types of the parameters, or the order of the parameters.

To overload an operator, C# and Kotlin support it syntactically with the `operator` keyword. In C# this keyword must be combined with the `public` and `static` modifiers. For Kotlin, we write it as `operator fun` (i.e., operator function). Arguably, Scala allows operator overloads with operator method declarations (e.g., `def +(other: Int)`). Java does not support operator overloading.

Type parameterisation is present in all selected languages for classes, interfaces (or traits), and methods. In C#, it is also applicable to structs and delegates. Every language supports variance, where the invariant does not require explicit syntax in addition to the generic type. Java makes use of wildcards to mark covariance / upper type bounds (`<? extends T>`), contravariance / lower type bounds (`<? super T>`) and bivariance (`<?>`). C# and Kotlin use `out` to mark covariance and `in` to mark contravariance. Scala uses `+A` to mark covariance and `-A` to mark contravariance.

In C# it is also possible to define constraints in addition to variance with the keyword `where`. It can be used for upper-type bounds, nullability, among other constraints [27]. In Scala, the lower bounds are set with `[A <: U]` and the upper bounds are set with `[A >: U]`. It is also possible to combine lower and upper bounds. In Kotlin, only upper bounds can be set, with `<T : U>`.

2.2.6 Functions

It is important to focus on several aspects of functions within the MP setting. We cover their citizenship within the type system of the language, lambda functions, higher-order functions, and nested functions.

Scala and Kotlin adopt the most functional approach to the inclusion of functions as types within their language. Functions can be used as a type (e.g., `f: Int => Int`). Java relies on *functional interfaces*, which are interfaces with a single abstract method (SAM), to apply a type to functions. C# designed `delegate` types to describe the signature of a function, which can be used as a type. Arguably, Java and C# do not support functions as first-class citizens because they are wrapped in an SAM interface. The Java compiler inserts a call to the single function on a SAM interface for lambda functions. The C# compiler inserts a call to the single function of the delegate[28].

In Java, anonymous classes can be instantiated using interfaces. Since Java 8, the lambda expression was introduced, which makes the instantiation of an SAM interface (e.g., functional interface) less verbose. Variables that are in scope and used by a lambda expression cannot be mutated inside or outside of the lambda scope. Under the hood, Java lambda expressions translate to these SAM interfaces, but they require far less syntax. The C# compiler similarly infers a delegate that is applicable to a lambda expression. In Scala, lambda functions are called anonymous functions. In Kotlin, lambda expressions are enclosed in curly brackets.

While Java prevents variables from being accessed from outside the lambda function, it allows for the modification of fields within the referenced object. C#, Scala, and Kotlin do not have this enforced protection against modification of external variables. Every mutable field accessible from the lambda expression can be mutated.

Every selected language supports higher-order functions to its full extent, due to the inclusion of functions as a type within their language. One could argue that functional interfaces in Java and Kotlin, and delegates in C# increase the readability of higher-order functions because the function is named. Naming those functions is also possible with type aliasing in Scala and Kotlin, but these aliases must be declared elsewhere, which might make them harder to understand.

Continuing with the readability aspect, nested functions are a common tool for dividing a function into readable chunks. In Java, this functionality is not integrated directly within the language, but via anonymous class declarations that implement a functional interface. This allows the developer to call the method via the anonymous class. C#, Kotlin and Scala have implemented local functions, which allow you to declare a nested function.

2.2.7 Referential transparency

An important principle of functional programming is referential transparency. We want to know how our selected languages enforce this principle, if at all. The languages are inherently object-oriented, which clashes quite hard with referential transparency due to the inclusion of state and the passing of state via functions.

In Java, C#, Scala, and Kotlin, the compiler does not allow us to check the enforcement of referential transparency on functions. This must be done by a developer inspecting the code or with a code analysis tool. In C#, we can, with the addition of the .NET Code Contracts library, use a `Pure` attribute on a class or function to specify that it is referential transparent. We cannot identify this as a language construct but rather as a language implementation.

2.2.8 Recursion

Our languages support head and tail recursion, since each language allows a call on the same function. It is more interesting to see how each language deals with the optimisation of recursive calls within their language, namely tail-call optimisation [24]. In Java and C#, the compiler does not support tail-call optimisation. Scala allows developers to annotate a function with `@tailrec` to signal to the Scala compiler that the function should be tail-call optimised. Kotlin allows the developer to add the `tailrec` modifier to a function declaration to specify that it should be tail-call optimised by the Kotlin compiler.

2.2.9 Lazy evaluation

All languages support lazy evaluation of values, but there is a major difference in how and to what extent lazy evaluation is implemented. In Java, lazy evaluation can be implemented with the `Supplier<T>` functional interface. Lazy evaluated values can also be created with the singleton pattern. The C# standard library also contains a `Lazy<T>` interface. Scala implemented the use of lazy evaluation within the language syntax with the `lazy` keyword. After the first execution of the value, the result is cached for later use. The Kotlin standard library contains the `Lazy<T>` interface and an instance of this interface can be returned with a delegated property called `lazy()`. This interface instance caches the result after the first execution, so subsequent calls simply return the cached result.

When we look at the similarities and differences in how lazy evaluation is handled by every language, we clearly see that only Scala supports it within its language syntax. We can argue that this is translated under the hood to a similar implementation as the other languages, but it clearly shows the focus on functional-styled features in Scala.

2.2.10 Pattern matching

Following the definition of FP pattern matching, it is a function that contains a set of possible equations, of which each is applicable in a certain situation. This function takes one argument, the subject. It contains two or more possible decisions that can be chosen on the basis of which of the decisions matches the subject. When analysing our MP languages, we see one type of construct that allows pattern matching. In Java 14 and C# 8, `switch` expressions were introduced. Scala and Kotlin included the `match` and `when` expression out of the box, respectively.

As shown in Table 2, C# and Scala support the most patterns that can be matched. Kotlin follows thereafter with support for some patterns and the possibility of creating more complex patterns from scratch. Java lacks pattern matching functionality with a small number of possible patterns. Java is working on adding more patterns, some of which are released as preview features of Java 17+[2]. Kotlin is currently developing a successor for its compiler, called the K2 compiler, which will enable many of the requested pattern matching functionality that is currently missing[3].

[2] `https://openjdk.org/jeps/406`

[3] `https://youtrack.jetbrains.com/issue/KT-186/Support-pattern-matching-with-complex-patterns`

Table 2: Pattern-matching features

	Java	**C#**	**Scala**	**Kotlin**
Default Pattern	Yes (`default`)	Yes (`default`)	Yes (`_`)	Yes (`else`)
Constant Pattern	Yes	Yes	Yes	Yes
Type Pattern	No[1]	Yes	Yes	Yes
Range Pattern	No	No	No	Yes (with `in`)
Guard Pattern	No[1]	Yes (with `when`)	Yes	Yes[2]
Declaration Pattern	No[1]	Yes	Yes	Yes
Property Pattern	No	Yes (with `is`)	No	No
Var Pattern	No	Yes	Yes	No
Positional Pattern	No	Yes	Yes	No
Sequence Pattern	No	No (Only C# >10)	Yes	No
Relational Pattern	No	Yes	No	Yes[2]
Logical Pattern	No	Yes	No	Yes

[1] Preview feature
[2] Pattern possible with expression(s)

2.2.11 Currying

Our programming languages provide different types of constructs to enable currying. From the definition in Section 2.1.2, we know that currying is applied when a multi-argument function is separated into multiple single-argument functions, e.g. `call(a,b,c)` becomes `call(a)(b)(c)`. In Java, we can chain methods as long as it is higher-order in the sense that it returns a functional interface, but syntactically it does not look like currying. In C#, currying can be established with delegates, although the C# compiler will transform the curried function into a method call chain with `.invoke()` methods. Kotlin handles currying similarly to C#, where a function needs to be returned to chain calls. Scala is the only language that supports currying out of the box. Methods can contain multiple parameter lists, allowing for partial application. Therefore, in Scala you can create curried functions.

2.2.12 Interfaces & Traits

Interfaces are an important part of OOP, defining a common set of functionality that classes can implement. Java, C#, and Kotlin use the term interface within their language, while Scala implements traits. In this thesis, we see traits as an interface, although they do have some extra features.

In Scala, a self-type (i.e., reference to another trait) can be defined within a trait to signal to classes that implement the trait to also implement the referenced trait. Traits can also have arguments to define properties at declaration level. Due to all these extra features, traits are more powerful than interfaces.

The features that an interface contains differ quite a bit between our languages, as shown in Table 3. Java is the only language that allows static methods inside an interface. Scala is the only language that allows for non-abstract mutable properties. Only in Java, it is impossible to define other access modifiers than `public` or `private`.

A common design principle within OOP languages is the "favouring of composition over inheritance" [26], which states that we must give priority to implementing interfaces instead of extending classes. A common drawback of this principle was the need for every derived class to implement abstract methods. It is interesting to see that all the selected languages have solved this drawback by implementing the delegation pattern with the introduction of default methods.

	Java	**C#**	**Scala**	**Kotlin**
Abstract method	Yes	Yes	Yes	Yes
Default method	Yes	Yes	Yes	Yes
Static method	Yes	No	No	No
Private method	Yes	Yes	Yes	Yes
Private static method	Yes	No	No	No
Abstract property	No	Yes	Yes	Yes
Non-abstract property	No	No	Yes	Yes* (only for `get`)
Extend another interface	Yes	Yes	Yes	Yes
Type parameterisation	Yes	Yes	Yes	Yes

TABLE 3: Interface Features

2.2.13 Records

A special kind of class that finds its place in all the languages that we have analysed is *record*. We must not confuse this class name with the record in Section 2.1. It is a class whose primary objective is to store and describe data in a syntactically concise manner. It removes a lot of boilerplate code by auto-generating getters & setters for properties and a default implementation of the `equals()` and `hashCode()` methods. Records are compared on property values, not by class reference. Records cannot be abstract and are always final.

Java 14 introduced the `record` that can only contain immutable properties, defined at the declaration in the record header. C# 9 introduces `records` which use value-based equality like Java records, but properties can be mutable. Scala implemented the `case class` out of the box to model immutable data. Kotlin uses the `data class` to model mutable and immutable data with auto-generated methods.

2.2.14 Sequence Comprehensions

Sequence comprehensions combine object-oriented data structures with functional features to iterate over large collections of data. An important property of sequence comprehensions is lazy execution, where the sequence comprehension is only executed when a terminal operation is called. Sequence comprehensions can be divided into three phases. We create a sequence, optionally define intermediate operations (e.g., with lambda functions) on the sequence, and we return a result (i.e. the terminal operation).

Every language supports a variation on sequence comprehensions, where Java & Kotlin support it mainly on implementation-level and C# & Scala support it at the syntactical level. In Java, we were introduced to sequence comprehension with *Streams*. Kotlin introduces us to *Sequences*. In C#, we are introduced to *LINQ* and Scala has *for-comprehensions*.

2.2.15 Conclusion

The analysis of these languages describes how these languages came about and how they are evolving. It shows a pattern on the integration of more functional concepts within these languages. Java being the oldest, it is apparent that they move the slowest due to their rigorous approach in updating their language, focus on backward compatibility, and their general reluctance to implement new features quickly. C# also has many interesting functional features within their language, with an ever-growing list of pattern matching functionality and LINQ being a strong functional querying mechanism. Scala and Kotlin were designed with functional style in mind, which is shown through their abundance in such features. Even though Scala and Kotlin can be brought back to Java bytecode, they are able to provide a functional style of programming on a syntactical level. Where Scala is the most functional-like language within the selection of languages, Kotlin is situated between Java and Scala.

Within the cross-comparison, a good distinction between language constructs and language implementations was regularly difficult to make. Where reserved keywords are (part of) the language's constructs, the same verdict could not always be made about SAM interfaces, that is, functional interfaces in Java and C#. Functions are not a concept on itself; they are an abstract notion of a function, but just an interface during compilation.

When abstracting features of these languages, we are often required to make a choice of what to see as language features and vice versa. It is important to determine the goal of this abstraction. What is measured with the language-agnostic representation makes the most important argument to include a construct. This goal will be covered in Section 3, and the selected constructs that benefit this goal are covered in Section 4.

With programming paradigm concepts of our selected languages covered in this section, it is interesting to see how major programming languages contain a combination of OOP and FP constructs and how they work together. Generally, selected languages make increasing use of pattern matching constructs, (lambda) functions, records, and sequence comprehension. These inherently functional concepts get intertwined with the mutable nature of OOP, creating new ways in which the developer needs to reason about the code.

2.3 Software Quality

With a better understanding of the constructs of the selected multi-paradigm programming languages and how they relate to each other, let us introduce the strategy for measuring the quality of software written in multi-paradigm code. In this section, we will describe the definition of software quality and its concepts.

Whenever there are humans in the loop, mistakes can be made. It is not different for software, where software can have catastrophic effects when it contains these human errors [4]. To minimise the risk of mistakes, software quality assurance (SQA) can be applied. SQA aims to assess the adequacy of software processes to establish confidence that the developed software is of good quality for its intended purposes [29].

There are benefits with the addition of SQA (in some form) within a software project. SQA aids prevention of mistakes, creates a quality assessment culture, streamlines the quality management process, and increases development progress. In short, it helps businesses measure the productivity of their software practices and provide guidance for improvement [30].

To standardise software quality and its concepts, ISO and IEC have created the ISO/IEC 25010:2011 standard [6], which improves on the ISO/IEC 9126:1991 standard [31]. This standard contains the product quality model. It focuses on the static properties of software and the dynamic properties of computer systems. The product quality model consists of eight characteristics, namely functional suitability, performance efficiency, usability, reliability, maintainability, portability, compatibility, and security.

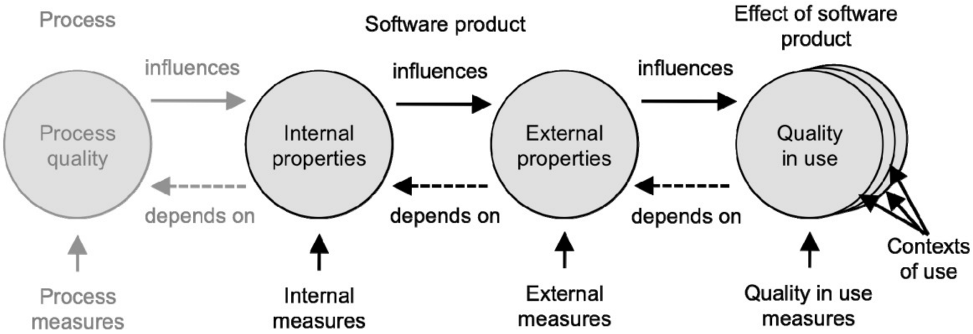


Figure 3: Quality in the lifecycle [6]

As shown in Figure 3, there are several quality concepts involved in quality assurance. The first independent part of measuring the quality of software is to determine the quality of the process, i.e., the development environment in which this software is created. In the second part of the lifecycle, the internal properties of a software product are evaluated. It is only logical that the process influences the internal properties of a system. Coding guidelines or project management approaches can influence the way code is developed. It is difficult to measure process quality in an automated manner, as we depend on external tools and manual input to extract process information. Within code quality measurement, we can use automated tools to extract important measurements. The parts after internal measures are also influenced by external factors, which might make it harder to automate. We are also focusing on language constructs of multi-paradigm languages, which are on the internal properties level. Therefore, we focus solely on the internal properties and its corresponding internal measures within the quality lifecycle.

2.3.1 Code Quality

The goal of code quality is to measure the structural quality of an application's source code. In simple terms, code quality is a good indicator of the human ability to reason with the source code and maintain it.

Code quality can be described with the following quality characteristics: maintainability, flexibility, portability, reusability, readability, testability, and understandability [32]. These characteristics are well suited for quantitative measurement by using metrics that tell something about one (or more) quality attributes. Only two characteristics are part of the core list of ISO/IEC 25010:2011, namely maintainability and portability. The other characteristics are subcharacteristics of maintainability and portability.

Although the portability characteristic is part of internal software quality, it says little about the source code itself, but more about the host operating environments on which it can run. Therefore, we choose only to include the maintainability characteristic and its subcharacteristics for measuring code quality.

When measuring maintainability, it is important to have a clear understanding of the term maintainability and also its subcharacteristics. *Maintainability* stands for the degree of effectiveness and efficiency with which the intended maintainers can modify a product or system. The subcharacteristics and their definitions are [6]:

- *Modularity*: degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components.
- *Reusability*: degree to which an asset can be used in more than one system or in building other assets.
- *Analysability*: degree of effectiveness and efficiency with which it is possible to assess the impact on a system of an intended change to one or more of its parts, diagnose a product for deficiencies or causes of failure, or identify parts to be modified.
- *Modifiability*: degree to which a system can be effectively and efficiently modified without introducing defects or degrading the quality of the existing product.
- *Testability*: degree of effectiveness and efficiency with which test criteria can be established for a system or component and tests can be performed to determine whether those criteria have been met.

2.3.2 Measuring Code Quality

To measure the quality of the source code, we can use metrics. The metrics tell something about a certain quantifiable or countable characteristic of software [33]. We can connect these metrics to the code quality characteristics that were covered in the previous section.

Heitlager et al. developed the Maintainability Model (MM) [3], which is a model to measure the maintainability of the source code. It is inspired by the Maintainability Index (MI) [34]. While MI outputs one "score" for maintainability, the MM sought to improve the evaluation of maintainability by creating a model with which root-cause analysis could be applied. The initial MM was later extended by SIG in collaboration with TÜViT [35]. This improved MM maps the subcharacteristics of maintainability from ISO25010 to eight independent system properties, as shown in Table 4. The properties are divided over four levels [35]:

- *System*: all software needed to achieve the overall functionality of the product.

- *Component*: a top-level subdivision of a system in which the source code modules are grouped based on a common trait.
- *Module*: a delimited group of declarations, i.e., classes in OO languages.
- *Unit*: the smallest named piece of executable code. This can be a method or a function literal.

The properties are contained in a set as small as possible, yet are technology-independent, easy to measure, and not strongly correlated with other metrics. The eight properties are [35, 36]:

- *Volume*: What is the size, in man years, of the codebase?
- *Duplication*: How many duplicated chunks of code exist in the codebase?
- *Unit size*: What is the size of every unit of code?
- *Unit complexity*: How many branch points does every unit of code have?
- *Unit interfacing*: How many parameters does every unit of code have?
- *Module coupling*: How many calls are made to a class from outside the class?
- *Component balance*: How well is the size, in man years, distributed between all components?
- *Component independence*: How many calls are made to a component from outside the component?

An important aspect of the SIG Maintainability Model is its ability to provide threshold values to adequately support decision making for systems under evaluation. Where other quality models lack information for quality improvement, threshold values allow SIG to objectively assess the quality of systems. These threshold values are derived with a data-driven, robust, and pragmatic method [37]. This method is used in 100 proprietary or open-source projects. The method focuses on defining threshold values with direct applicability to differentiate software systems, judge quality, and pinpoint problems. With the threshold values known, we can apply a rating for each system property in a software system. This rating is on a scale of 1 to 5 stars. It is derived from the distribution of the metric results with respect to the threshold values. For example, a higher percentage of unit complexity above the threshold value will result in a lower rating. In Table 4, we can see how each property of the system is mapped to a quality attribute. Using this mapping and the rating for each property, we can calculate the maintainability rating.

Research by Luijten et al. [38] found empirical evidence on all three levels of the SIG Maintainability Model that systems with a higher maintainability score have a higher problem-solving efficiency. Only *unit interfacing* had no correlation with the resolution time of defects or enhancements.

	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling	Component balance	Component independence
Analyzability	X	X	X				X	
Modifiability		X		X		X		
Testability	X			X				X
Modularity						X	X	X
Reusability			X		X			

Table 4: Relation between subcharacteristics and system properties [36]

2.3.3 Multi-Paradigm Code Quality

In academic research, the multi-paradigm aspect of code quality is not widely researched. This gap is partly filled by previous theses on fault proneness within individual multi-paradigm programming languages. Landkroon created a logistic regression prediction model to find the fault proneness of Scala source code [11]. It uses both OOP metrics by Chidamber & Kemerer [7], FP metrics [10], and an issue tracker with 'faulty code' classifications to feed into the prediction model. He states that code quality is a vague concept, which is why he chose to quantify the quality of code via fault proneness of classes.

In Zuilhof's research, a similar type of fault proneness prediction is performed, but for the programming language C#[12]. He states that FP inspired constructs, when unknown in both OOP and FP, introduce a new problem space and a new kind of complexity. The research focuses on using metrics that highlight code complexity and lambda function purity. He proposes several MP metrics, which are proven to strongly correlate in the context of fault proneness detection. There was a correlation between the candidate metrics and error-proneness, but the presence of this correlation is too uncertain to make claims about causality. The candidate metrics by Zuilhof can be seen as anti-patterns leading to an increased amount of faults. These anti-patterns are "side effects in lambda expression", "complex lambda expressions", and "side effects i.c.w. lazy evaluation".

The research by Konings applied the candidate metrics by Zuilhof in Scala, including some new metrics tailored specifically to Scala constructs to improve the baseline model by Landkroon. Konings uses compiler trees to create ASTs that include type information. It is mentioned that not all sources (libraries and external sources) are always available and therefore the typing information might be incomplete. The compiler trees also desugar some source code, which can bias the analysis between the real and the desugared source code. For Scala, the MP candidate metrics proposed by Zuilhof did not significantly improve the fault detection performance of the baseline model. It did contribute by indicating why the code contained faults, which is important for root-cause analysis.

2.4 Compilers & Metaprogramming

We have provided several insights into programming paradigms, languages, constructs, and code quality. To measure code quality at a language-agnostic level, we must be able to extract the information we need for such an analysis. To extract that information, we can reuse and get inspired by techniques used by language compilers to process source code.

For software to work, it needs to be processed from human-readable source code to machine-readable code. This is called compilation, which is done by a language compiler. Every language requires one to create executable code from source code. During the compilation process, there are several steps that must be taken to create an intermediate form of the source code. In the following sections, these phases are covered.

2.4.1 Source Code

Source code is human-readable text that defines a set of instructions for what a software program can do. Our definition of source code includes all physical lines of code, including lines of logical code, comments, and blank lines [39]. All of these lines are important for the user who wants to understand the source code structure and domain-specific context, but only lines of code that can be run will be compiled.

2.4.2 Symbol Table

The symbol table is at the centre of the compilation process and is used in all phases of the process. It is a data structure in which information about declarations, scopes, and type information is stored. It is used in the compilation process to make it go faster and to ensure that the semantic information contained in the intermediate representations is correct.

2.4.3 Control Flow Graph

A control flow graph (CFG) is a way in which the flow of a software program can be represented. It is a graph-based representation of control flow relationships within a software program. The technique was created by Frances E. Allen in 1970 [40]. It is a widely used technique for static analysis of a program as it can accurately describe the flow of (part of) a program.

2.4.4 Lexical Analysis

During lexical analysis, source code is translated into a sequence of lexical tokens. This is done within the compiler using a lexer. A lexer can also be called a tokenizer or scanner. A lexer takes a stream of characters and turns them into lexemes or tokens. These lexemes can be identifiers, keywords, operators, literals, comment lines, etc. [41].

2.4.5 Syntax Analysis

The token stream from the lexical analysis is fed into the syntax analysis, which is also called "parsing". A concrete syntax tree (CST) (i.e., parse tree) is a mapping of a language grammar to a tree form [41]. During syntax analysis, the CST is used to efficiently check whether the syntactic structure of the token stream is correct.

2.4.6 Semantic Analysis

To create the abstract syntax tree (AST), the CST is populated with the tokens of the token stream. Some token information is discarded because of its lack of importance for analysis purposes, while it might have had value for parsing purposes. During semantic analysis, the semantic correctness of the CST is checked using the symbol table, and the source code is checked to see if it is semantically consistent with its language definition (i.e., grammar) [41].

2.4.7 Metaprogramming

These intermediate products become interesting when we measure the quality of the source code. A technique called metaprogramming can be used to interact with one or more of the intermediate representations. With metaprogramming, we can manipulate other programs and also perform an analysis on them [42].

There are many tools available for metaprogramming. Metaprogramming tools help us analyse source code, so it is helpful to learn more about these libraries and use them to get inspired for design choices in our language-agnostic framework.

Rascal
Rascal is a well-maintained metaprogramming language capable of analysing, transforming, and generating primitives of source code at the language level [43]. It is also useful for creating CSTs for programming languages. These CSTs can be traversed with a built-in Rascal library.

ANTLR4
ANTLR (ANother Tool for Language Recognition) is a powerful parser generator to read, process, execute, or translate structured text or binary files [44]. It is widely used to build languages, tools, and frameworks. ANTLR provides a long list of grammar specifications for programming languages on its GitHub page[4]. From a grammar, ANTLR generates a parser that can build CSTs from a source code. ANTLR can be used as the primary step in transforming source code into an AST [45].

Roslyn
C#wanted to change the way compilers generate machine code by transforming the compiler into a platform with APIs. Developers can request all kinds of information from the compiler. It is known as the .NET Compiler Platform SDK, also known as Roslyn [46]. With it, developers can retrieve all the wealth of information the compiler has about the source code. Thus, this SDK dramatically reduces the entry barrier for creating code-focused tools and applications. This SDK is an extension that can be installed using Visual Studio. It is possible to explore how code is represented in a syntax tree using the Syntax Visualiser in Visual Studio. You can use the parse methods of `CSharpSyntaxTree` to parse text into a C#syntax tree.

Scalameta
Scalameta is a library for Scala projects that can parse source code into syntax trees. In contrast to Roslyn, Scalameta is an external project. It enables the developer to read, analyse, transform, and generate Scala programs at a level of abstraction [47]. Scalameta

[4] `https://github.com/antlr/grammars-v4`

trees are lossless and therefore are great for fine-grained analysis of source code. An AST explorer[5] was built with Scalameta to analyse the AST of Scala files.

Kotlin Symbol Processing

The Kotlin Symbol Processing API, developed by Google, is a library for performing high-level metaprogramming using idiomatic Kotlin. Compared to other Kotlin compiler plugins, it is not very heavy because it only stores information about types at the declaration level. All information on expression level is not examined, which is also not a goal of this API. Therefore, this library is useful for analysing type information but not for analysing the entire source code.

Kotlinx.AST

Kotlinx.AST is a parsing library for Kotlin source code. It uses ANTLR4[6] with the official Kotlin grammar [48]. This library is in the early stages of development and supports only JVM as the target language.

UAST for JVM languages

The Unified Abstract Syntax Tree (UAST) is an abstraction layer that sits on top of the Program Structure Interface (PSI) for JVM languages. It is part of the JetBrains IDE toolkit for developing IDE-specific plugins. It gives a common API for working on the different parts of the JVM language in the same way. It currently fully supports Java, Kotlin, and Scala. Because there is no guarantee of the preservation of the ancestor-descendant relationship, a mismatch between the specific language structure and the UAST structure is possible.

[5] `https://github.com/fkling/astexplorer`

[6] `https://github.com/Strumenta/antlr-kotlin`

2.5 Code Quality Metrics & Smells

Section 2.3 covered the terminology for software quality, code quality, and its quality characteristics. In that section, we covered the SIG maintainability model and their language-independent terminology (i.e., project, component, module, and unit). In the following sections, this terminology will be used.

With the intermediate representations covered in Section 2.4, we can go into detail about the metrics that are used to measure code quality. We cover what the requirement is for each metric, how they are extracted from an intermediate representation, what their impact is on code quality characteristics, and what the recommended threshold values are.

Not all code metrics are related to a specific paradigm. On the most basic level, all programming languages consist of lines of code, and comment lines can be placed to explain that code. SIG has defined the maintainability model, a suite of eight characteristics with which the source code can be graded on maintainability [3]. For OOP, a well-known set of metrics was formally defined by Chidamber & Kemerer [7]. In a mapping study by Nuñez-Varela et al. [49], these metrics were found to be used by the majority of papers on the measurement of the quality of the OOP code. Another renowned OOP metric suite is that of Lorenz & Kidd. In FP, a Haskell metric suite was created that was inspired by the CK metric suite [10].

2.5.1 Lines of Code (LOC)

Lines of code is one of the most straightforward ways to measure the size of source code. This metric counts the number of logical lines of code, thus excluding comment lines and blank lines [50]. Complexity is said to increase in order of magnitude; therefore, 10K lines of code are assumed to be much less complex than 1M lines of code. Useful LOC metric aggregations are at the unit, module, component, and project levels. One way to count lines of code is to iterate over plain text with regular expressions. Another way is to use positional information from the concrete or abstract syntax tree.

The influence of LOC on quality varies depending on the language under consideration. Depending on the programming language, we can calculate the mean time it takes to create 1000 lines of code.

2.5.2 Cyclomatic Complexity (CC)

McCabe's Cyclomatic Complexity is one of the best known source code metrics. It measures the complexity of a unit based on the number of branch points within the control flow graph (CFG) of the unit. The equation of CC is shown in Eq. 1, where E is the number of edges in the CFG, N is the number of nodes that contain only one transfer of control, and P is the number of disconnected parts in the flow graph. CC can also be calculated from an AST, where an increment in the metric is performed for every if, ternary, while, do while, for, foreach, switch, case, `||`, `&&`, catch, and null propagation.

$$CC = E - N + 2P \tag{1}$$

2.5.3 Cognitive Complexity (COCO)

SonarSource, most famous for their SonarQube software analysis tool, developed an improvement on McCabe's Cyclomatic Complexity metric called the Cognitive Complexity metric [51]. They say that CC is no longer all-inclusive because it doesn't include try/catch and lambda functions, which are common in modern languages. Cognitive complexity takes into account these features of the modern language. The metric contains three aspects to measure cognitive complexity: control flow expressions or statements, nesting level, and nesting increments.

2.5.4 Weighted Method per Class (WMC)

This metric measures the complexity of a module. In Equation 2, we consider a class C with methods $m_1, ..., m_n$, where $c_1, ..., c_n$ be the Cyclomatic Complexity (CC) of the methods. This metric only considers direct declarations in a module, thus ignoring nested functions. Like CC, a higher value of this metric means that the class is more complicated. This metric can be used to catch large classes, where CC results could miss the sum of all functions within a module.

$$WMC = \sum_{i=1}^{n} c_i \tag{2}$$

2.5.5 Cognitively Weighted Method per Class (CWMC)

Similarly to Weighted Method per Class (WMC), CWMC sums the Cognitive Complexity (COCO) of the methods declared within a module. This metric is an adaptation of WMC but was not explicitly defined in the paper on cognitive complexity by SonarSource [51].

2.5.6 Depth of Inheritance Tree (DIT)

This metric describes the depth of the highest ancestor in the dependency tree of a specific module. It is assumed that a larger depth implies more reuse through inheritance and therefore greater complexity [7].

2.5.7 Number of Children (NOC)

The Number of Children metric counts the number of classes directly derived from a given class [7]. Module is used throughout this document, but the NOC metric is only applicable to non-final classes. The NOC metric shows how much a project reuses code through inheritance, which can show how much testing needs to be done.

2.5.8 Coupling Between Objects (CBO)

The CBO metric counts, for a class, the number of other classes to which the class is coupled by a call to a method or the access of a field. It specifies the number of classes on which one class depends within the project, including external libraries.

When the CBO is high, it is assumed that there is less modularity, which means it is harder to reuse the class. High CBO is said to decrease maintainability and testability [7].

2.5.9 Response for a Class (RFC)

The RFC metric is the number of methods that can be executed in response to a message received by an object of a class, known as the response set RS. To calculate RS, we use Equation 3, where R_i = set of methods called by method i and M = set of all methods in the class.

$$RFC = |M \cup \{\bigcup_{i=1}^{n} R_i | i \in M\}| \tag{3}$$

It describes the count of the number of methods in a class plus the unique method invocations to other classes that occur from those methods. The larger the number of methods invoked in a class, the greater the complexity of the class and, therefore, the testing effort [7].

2.5.10 Lack of Cohesion in Methods (LCOM)

The LCOM metric proposed in the CK metric suite looks at the degree of similarity between the variables used within the methods. It is a count of pairs of methods where the similarity between variables used is zero minus the count of pairs of methods whose similarity is not zero, shown in Equation 4, where I represents the set of variables used by method i.

$$\begin{aligned} P &= \{(I_i, I_j | I_i \cap I_j = \emptyset\} \\ Q &= \{(I_i, I_j | I_i \cap I_j \neq \emptyset\} \\ LCOM &= |P| - |Q| \end{aligned} \tag{4}$$

Low cohesiveness is indicative of poor design and high complexity. It has also been observed to indicate a significant probability of making mistakes. Ideally, the class should be divided into two or more smaller classes [7].

2.5.11 Parameter Count (PC)

PC is the active metric for measuring SIG's unit interfacing [3]. It counts the number of parameters belonging to a unit.

2.5.12 Number of Units (NOU)

NOU is the active metric to measure the size of the SIG module interface [3]. It counts the number of units that belong to a module.

2.5.13 Lambda Count (LC)

Lambda Count, originally proposed by Zuilhof as "Number of Lambda Functions Used In A Class" [12], is a metric to measure the amount of lambda use within a module. This metric is a primary indicator of the degree of functionally-styled code within a module.

2.5.14 Lines of Lambda (LOL)

While lambdas are very convenient and compact for injecting behaviour without having to create a unit or module, they can become too large. SonarQube has defined the Large Lambda code smell which describes that a lambda function is being misused and needs to be refactored (e.g., the "Extract Method" refactor)[7]. To detect this code smell, the lines of code encapsulated by the lambda function are counted. There is no available information that describes the risk thresholds for the number of lines.

2.5.15 Lambda Function with Side Effects (LSE)

LSE is a code smell, inspired by metrics from [12], that detects whether a lambda function mutates state outside the lambda function. Lambda functions are often not called in the

[7] `https://rules.sonarsource.com/kotlin/type/Code%20Smell/RSPEC-5612`

order in which they are declared. Hence, the developer must be cautious when defining lambda functions that contain mutations to the outer scope. It is not always clear what the state of the application is when the lambda function is actually executed.

Side effects can be detected at the AST level by checking whether assignments are made to fields outside the lambda function or if an impure unit is called. Detecting if a unit is pure is a complex undertaking. Österberg has defined several levels of purity that can be detected on an AST level, but it highlights that detecting impurity for external libraries will require the AST of the external library [52]. For large projects with many external libraries, this will be very slow and unpractical.

2.5.16 Length of Message Chain (LMC)

Message Chaining is considered a code smell within OOP [53]. It describes consecutive calls (e.g., `callA(1).callB(2).callC(3)`). Within FP, currying and partial application are similar to message chains. The mere difference lies within the return value of a unit, which is a module instance in the case of message chains, and a lambda function in FP.

When the message chain smell is used, the developer depends on navigating the module structure. It places a cognitive load on the developer. The cognitive load when using curried functions can be even higher, as there is no unit signature to describe the behaviour that is called. Therefore, we regard currying the same as message chaining in our multi-paradigm setting, which makes it a candidate MP code smell.

2.5.17 Depth of Looping (DOL)

The DOL metric counts the maximum number of consecutive loops that occur within a unit. As shown with the metrics CC and COCO, loops add complexity, and nested loops even more. Therefore, detection of depth of looping can point to places in the source code that require refactoring.

As covered in Section 2.2.14, sequence comprehensions can be a powerful tool for decreasing complexity when iterating through collections. In object-oriented languages, iterations are mostly done via loops. Therefore, in this thesis, we introduce Lack of Sequence Comprehensions (LSC) as code smell. LSC can be detected using the DOL metric.

2.5.18 Duplicate Code (DUP)

Duplicated code is almost always bad for the quality of the code because it adds redundant lines of code to the source code. Duplicated code makes the source code more difficult to understand and maintain. There are several types of duplication, namely exact clones (Type 1), parameterised clones (Type 2), near miss clones (Type 3), and semantic clones (Type 4) [54]. According to SIG, 6 lines of code are considered duplicates [3] (Type 1). Code duplication can be mitigated by applying refactoring (e.g., "Extract Method" [53]) to both parts of the duplication.

2.5.19 Metric Thresholds

A code metric itself is not enough information to act upon and drive a change in code quality. To pinpoint parts within the code that require extra attention, it is required to know at what threshold(s) a measurement is a quality risk. Determining such thresholds is a complex topic. Qualitatively, we can consult experts for their opinions about thresholds. Quantitatively, we can analyse the statistical properties of the source code metrics by analysing the distributions of the metrics [37, 55]. Such quantitative research analyses metrics computations and places them in percentile ranges from 70% for low risk, 80% for medium risk, 90% for high risk, and everything above a very high risk.

Even with thresholds derived from large data sets, an individual project can still be designed differently, with a different architecture, different code style, all of which will impact the underlying metrics.

		Low	**Medium**	**High**	
LOC	Unit Lines of Code	30	44	77	[37]
CC	Cyclomatic Complexity	6	8	14	[37]
DIT	Depth of Inheritance Tree	2	3	4	[55]
LCOM	Lack of Cohesion in Methods	> 0			[7]
PC	Parameter Count	2	3	4	[37]
NOU	Number of Units	29	42	73	[37]

Table 5: Available Metric Thresholds

2.5.20 Software Quality Tools

There are several tools available to measure the quality of software. In this section, we give a short description of some well-known tools. These tools can be used as inspiration for computing language-agnostic metric computations. These tools also serve as related work, where they can be used as benchmarks for our language-agnostic metric computations.

SonarQube
The SonarQube[8] framework is a Code Quality and Code Security tool that is built for easy integration into IDEs and CI/CD pipelines. It supports 29 programming languages for static analysis at the time of writing. Analyses run with SonarQube produce extensive reports that include various metrics such as complexity, code duplication, and technical debt and can help the developer by suggesting code quality improvements to their source code. When performing an analysis on a codebase, every file of which the file type is supported by SonarQube will be analysed. SonarQube allows developers to create their own plugins within the SonarQube framework. The developer is limited in their freedom to implement a plugin, since the libraries provided by SonarQube must be extended.

NDepend / JArchitect
NDepend is a software quality assurance tool for C#. It has a Java version called JArchitect. It can be used as a SonarQube plugin or as a software application. It can be used to query over C#code, show code dependency graphs, perform technical debt estimation, monitor the health of an application, and compute code metrics at the application, assembly, namespace, type, method, and field level.

Designite
Designite is a software design quality assessment tool for C#and Java [56]. It provides a command-line tool for analysing projects. It can detect architecture smells, design smells, implementation smells, and compute code metrics such as LOC, WMC, NOC, DIT, LCOM, CC, etc.

[8] https://www.sonarqube.org/

CPD

CPD (Copy/Paste Detector) is a code duplication detection tool. CPD has language support for Java, C#, Kotlin, and Scala. It uses ANTLR to tokenise the language for detection. It uses the Karp-Rabin String Matching algorithm for efficient duplication detection [57].

3 Code Quality Assurance

In previous sections, the quality characteristics for measuring code quality have been covered including code metrics that are required to be computed to be able to reason with the code in a concise manner. This background information is required because it provides guidance for the following sections. The primary research question of this thesis is to explore to what extent a code quality assurance framework can be established. For such a framework to be designed, it must be crystal clear what quality needs to be assured and how it will be assured. In this section, the Goal Question Metric approach is explained, and an implementation is described that guides the design of the code quality assurance framework.

3.1 The Goal Question Metric (GQM) approach

With knowledge of the selected code quality characteristics, it is important to formalise an approach to connect our goal of evaluating and improving code quality with metrics. To create such a mapping, the Goal Question Metric (GQM) approach can be used. GQM is a measurement model with the objective of providing a mechanism to define and interpret operational and measurable software [58]. It is made up of three levels:

- *Goal*: The first level is the conceptual level, i.e. the goal. The goal is defined for a product (e.g., specifications, designs, programs), process (e.g., designing, testing, interviewing), or resource (e.g., software, hardware, personnel).
- *Question*: The second level is the operational level, i.e. the question. Questions are used to characterise the achievement of a goal with respect to a selected quality issue and to determine the quality of a goal from a selected viewpoint.
- *Metric*: The third level is the quantitative level, i.e. the metric. It embodies the data that are related to the questions in order to answer them quantitatively. These data can be objective or subjective to a particular viewpoint from which it was taken.

3.2 Our Goal

As mentioned in our introduction, having a grasp of software quality helps minimise bad code from reaching production. In Section 2.3.1, we focused on measuring the maintainability of software. An integral part of maintainability is the developer. According to Lehman's laws, software evolves over time, and if it is not properly maintained, it will become increasingly hard to work with [59]. Research has shown that poor maintainability increases fault proneness and decreases developer productivity [60, 61]. With our focus on multi-paradigm software, it is yet as important to be aware of the software's quality and be able to make correct assessment keeping the multi-paradigm context in mind. Therefore, using the terminology of Section 2.3.2, the goal has been defined.

> **Analyse and evaluate mature multi-paradigm software projects with respect to maintainability of source code from the developer's point of view**.

Before continuing with the questions and metrics part of the GQM approach, let us dissect this goal. We only want mature projects that have a suitable size and are actively maintained. These types of mature project will benefit most from measuring maintainability because it will drastically impact the evolution of the project. To be able to correctly evaluate these projects, they need to be analysed from the point of view of a developer. This will influence the reasoning and prioritisation for the design of the framework.

3.3 Questions & Metrics

In line with our goal, we define several questions that need to be answered to assess each subcharacteristic of the maintainability characteristic. The first five questions have been extracted from the Maintainability Model by SIG [3] and reformulated to better suit our terminology. The sixth question has been formulated to capture the combination of OOP and FP within our framework.

On the quantitative level, it is important to have metrics that are not correlated, as this could negatively influence the outcome of our maintainability assessment. Using language-agnostic metrics created by SIG [35], OO metrics [7], MP metrics by Zuilhof [12], the literature on code smells [61, 62, 63], and the addition of Cognitive Complexity [51], we can answer each question at the operational level of our GQM model.

- **How easily can the code be built upon?**
  To capture the subcharacteristic of modularity, we challenge the ability of a developer to extend the source code.
  - Module lines of code (MLOC)
  - Coupling Between Objects (CBO)
  - Lack of Cohesion of Methods (LCOM)
- **How easy is it for the code to be reused?**
  Reusability is an important part of maintainable software. The "Don't Repeat Yourself (DRY)" principle can attest to that.
  - Unit Lines of Code (ULOC)
  - Unit parameter count (PC)
  - Depth of Inheritance Tree (DIT)
  - Number of Children (NOC)
- **How easy is it for someone else to identify the parts that need to be modified or to diagnose the code for deficiencies?**
  Where other subcharacteristics of the code are focused mainly on describing the internal design of the source code, this question focuses on the analysis of the source code. How easy is it for the developer to reason with the source code and find the parts of the source code that introduced a fault or a code smell?
  - Module Lines of Code (MLOC)
  - Project 6 line duplicates (DUP)
  - Response for a Class (RFC)
- **How easy is it to make adaptations to the code?**
  Once source code has been identified to require modifications, it is important to see how modifiable the source code is. For example, tightly coupled source code will be more difficult to modify.
  - Project 6 line duplicates (DUP)

  - Cyclomatic Complexity (CC)
  - Cognitive Complexity (COCO)
  - Weighted Methods per Class (WMC)
  - Cognitively Weighted Methods per Class (CWMC)
  - Coupling Between Objects (CBO)
  - Lack of Cohesion of Methods (LCOM)

- **How easy can the code be tested?**
  Tests within a software project, such as a software quality assessment framework, help the developer minimise the number of faults that reach production. It is important to be able to write tests easily because it will increase maintainability.

  - Module Lines of Code (MLOC)
  - Unit Lines of Code (ULOC)
  - Cyclomatic Complexity (CC)
  - Cognitive Complexity (COCO)
  - Coupling Between Objects (CBO)
  - Weighted Methods per Class (WMC)
  - Module Lack of Cohesion of Methods (LCOM)

- **How well is the structural design of the code conforming to multi-paradigm?**
  Quality software is built following design patterns made with the expert knowledge of the developer. Code smells can affect different aspects of code quality, such as analysability and modifiability, and could lead to the introduction of faults [53]. We argue that the integration of multiple paradigms introduces minor changes in the OOP and FP design patterns. Several metrics have been defined to answer the question:

  - Lambda Count (LC)
  - Lambda Lines of Code (LLOC)
  - Lambda Function with Side Effects (LSE)
  - Length of Message Chain (LMC)
  - Depth of Looping (DOL)

4 LAMP Metamodel

In Section 2.2, the four MP languages were analysed and compared. In Section 2.5, we gathered a set of metrics that can be used to assess the quality of the source code. It is not yet clear how this can be done on a language-agnostic level. For that reason, this section describes a language-agnostic multi-paradigm representation, namely the LAMP metamodel.

Designing a model is fairly simple, but designing a correct model requires much more thought. To provide an initial structure for the model, we would use the grammars of our MP languages, which provide the blueprint for the AST of each language.

With the definition of the metrics, we could design parts of the model that can compute each metric. Finding general-purpose abstractions within language constructs is a time-consuming task that requires extensive knowledge about every language grammar, which is why the features of these languages were analysed in the first place. Appropriate names for each model element must be found that capture the meaning behind the abstraction. We must also take language-specific context into account, since structural information from the language could be required for other kinds of analysis, e.g., code smell detection or detecting duplicates.

With these challenges in mind, the choice was made to design this metamodel iteratively. From each iteration, we could learn how the model could represent the language constructs and if it was clear to understand. In total, we did four major versions of the metamodel and eventually decided to implement the third version. Although a complete description of this thought process is provided in Appendix B, it is important to give a brief summary of each iteration to give a sense of why certain choices were made.

To reduce the cognitive load due to the immense size of the grammar structure of multi-paradigm languages, the choice was made to stay close to the Java grammar in the first version. Java was the only language that provided documentation for its grammar, compared to the other selected languages, which was helpful in gaining a grasp of the entire structure. In the second version, we distil five major elements that were part of every grammar: scope, module, property, unit, and expression. Every element within the metamodel had to be seen as scopes or building blocks. In the third version, improvements focused on terminology, the integration of scope inside the main elements instead of a separate element, and the addition of subtypes of Expression to enable the computation of some important MP metrics. Finally, a fourth version was created that removed many of the previously defined elements from the metamodel. It served as a "minimal" acceptable version of the elements that needed to be in the metamodel to make it work. The third version was eventually chosen for implementation due to its expressiveness within the modelled elements, which was partly gone in the fourth version.

Therefore, we mark the third version of our metamodel as the "final" version. Every element of this metamodel will be covered in the following subsections. The design choices will be discussed as well as the construct to which each element relates, and which metric requires its presence within the metamodel.

4.1 Overview

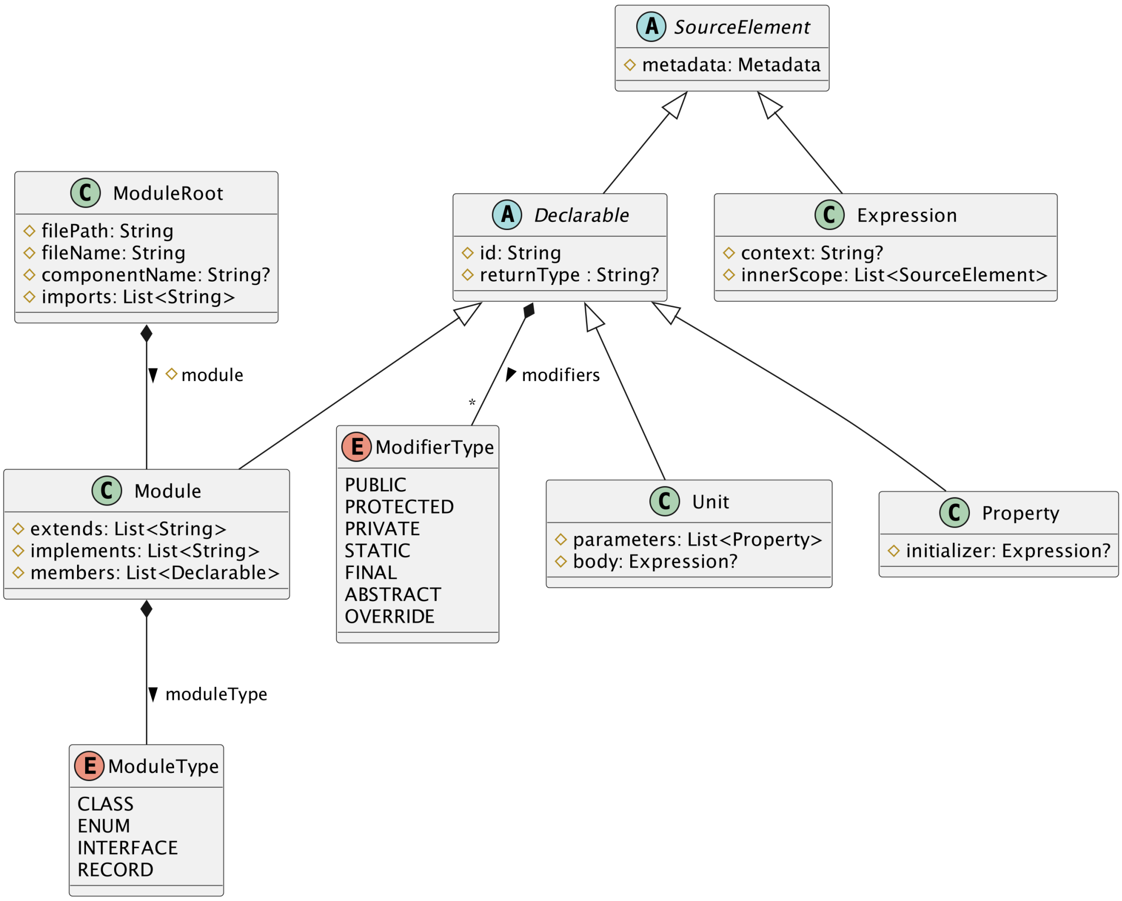


Figure 4: LAMP Metamodel - Overview

The LAMP metamodel conforms to a tree structure similar to abstract syntax trees. ModuleRoot, as shown in Figure 4, is the root node of our tree and contains the location and name of the source file, the name of the component, and imports within the source file.

Module is the child of this root node and can contain child nodes, i.e., members, which are of type Declarable. A member can be a module, unit, or property. Declarable is a subtype of SourceElement. SourceElement is the main building block that can be placed as a node within the tree, either as a subtype of Declarable or as a subtype of Expression.

In the following subsections, every element within the metamodel will be covered. Each element has been thoughtfully modelled to incorporate important aspects of multi-paradigm constructs. The model relies heavily on subtyping to reduce redundancy of elements within the metamodel. To keep the figures concise, the figures for each element only show the direct subtypes and immediate children of the element.

4.2 ModuleRoot

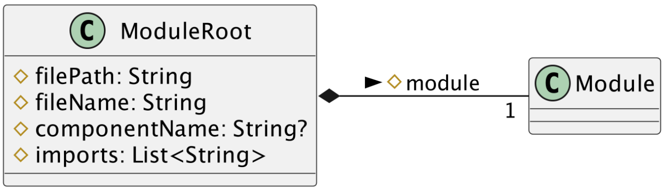


Figure 5: LAMP Metamodel - ModuleRoot

The ModuleRoot is the entry point of the metamodel; it is the all-knowing element that glues the elements together. At the root level, a class, interface, enum, or record is situated under the collective term Module.

To facilitate a reference to the actual class file in a file system, the absolute file path and file name are added as properties to a ModuleRoot. This is the information required to facilitate suggestions for improvement based on the metrics computed.

Class files are generally structured by defining the component name (e.g. as package or namespace) and a list of imports. Although present in most applications, classes can be declared without a component. When present, we refer to the component by its full name (e.g. `"nl.utwente.thisisacomponent"`). Imports appear similar to component references, but when referencing all content of the component, an asterisk (i.e. `".*"`) is put as the suffix of the reference. These naming conventions are used to be able to traverse components in the inheritance tree in a uniform manner.

4.3 SourceElement

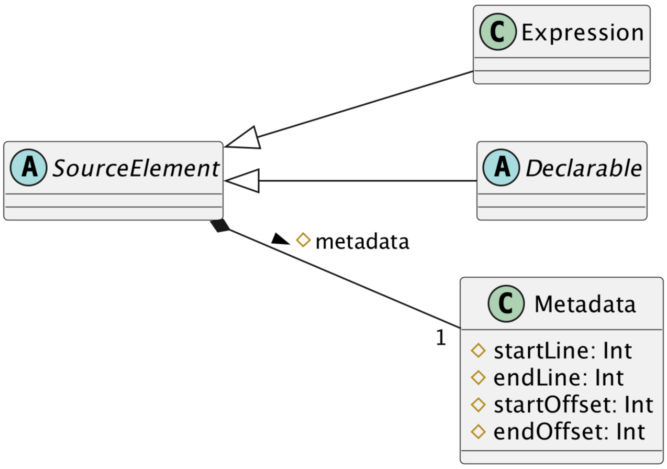


Figure 6: LAMP Metamodel - SourceElement

Every element in the LAMP metamodel, except ModuleRoot, is a subtype of SourceElement. This element describes every element that is part of the source code. It stores the exact code position of this element as Metadata. Metadata contains the starting and ending line of the source element, as well as the start and end offset. Every element that extends SourceElement, therefore, inherits this positional metadata.

The metadata of an element is required for the LOC metrics to be computed. It also enables us to point to the exact location of a source code element within a module.

4.4 Declarable

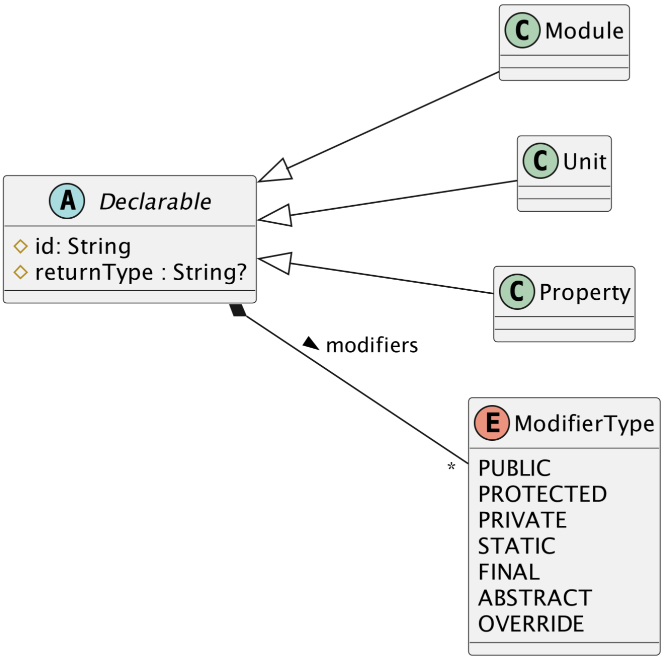


FIGURE 7: LAMP Metamodel - Declarable

The abstract Declarable class describes all types that can be declared within a module or expression. It contains an identifier that is used to reference the declarable type from other parts of the code, which will be covered in detail in Section 4.8.6.

When a declarable type is referred to from another part of the metamodel via an Access, it is important to know which type is returned for type resolution purposes. For example, the access chain `getAccount().balance` is modelled, and we need to know what the control flow is of the individual access references. The `getAccount()` unit call (subtype of Access) refers to a unit. This unit contains the return type `Account` which signals that the following access in the chain refers to a declarable in `Account`. Therefore, the return type of `balance` can be resolved by consulting the declarable `Account` and finding the property with id "balance". Without the return type, it is impossible to get this information about the control flow that spans multiple modules.

A declarable also contains zero or more modifiers. As covered in Section 2.2.1, there are many modifiers available in the selected languages. There are access modifiers (such as `public`, `protected`, and `private`) as well as non-access modifiers (such as `final`, `static`, `abstract`, and `override`). To efficiently query within the metamodel, it is not necessary to make the difference in categories of modifiers explicit.

Taking into account the code metric requirements, there are a few modifiers required for a correct computation of the metrics using the metamodel. Most of the time, access modifiers are required for finding a referenced declaration. Some metrics (public unit count, overridden unit count, etc.) also need a modifier to check if a declarable contains that modifier.

We must be mindful about transforming languages to the metamodel when copying access modifiers. Modifiers may need to be added by hand to individual declarable types when using the Module element. For example, in Kotlin, every class is final without the syntactic presence of a `final` keyword. Therefore, this explicitness regarding the internal behaviour (or absence) of modifiers requires additions of modifiers when transforming such behaviour.

There are several modifiers that have significant value for computing metrics or resolving type information. Consequently, the following modifiers are included in the metamodel:

- `PUBLIC`: Declarable is publicly available within the project.
- `PROTECTED`: Declarable is accessible from the inheritance tree and the same component.
- `PRIVATE`: Declarable is available within the same ModuleRoot.
- `STATIC`: Declarable is bound to the Declarable type, not the Declarable instance.
- `FINAL`: Declarable is immutable.
- `ABSTRACT`: Declarable is abstract.
- `OVERRIDE`: Declarable overrides a parent Declarable.

4.5 Module

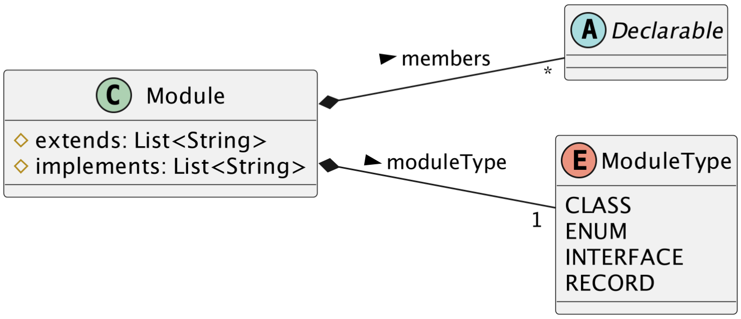


FIGURE 8: LAMP Metamodel - Module

Module is the language-independent name for a class type, as proposed by Heitlager et al. [3]. During the analysis of the selected languages, a common structure was found between class, enum, interface, and record. Each of them implements properties, units, or inner modules to some extent. For that reason, the element contains members that extend the abstract Declarable type.

Within Module, the "extends" and "implements" properties are defined as lists of strings. In regular classes, enums, and records, extensions are limited to a maximum of one. This behaviour is different for interfaces, where an infinite number of other interfaces can be extended. Therefore, the "extends" property is modelled as a list of strings instead of a string. A string represents an id that is the reference to the parent Module. For the "implements" property, an infinite number of interfaces can be implemented on a Module. For that reason, this property also refers to a list of strings that reference each interface Module.

In Section 2.2, we cover the differences between classes and objects. Objects offer encapsulation of static members of a class in a concise and developer-friendly way. Therefore, objects can be seen as static classes with static members and thus are seen as such within the metamodel.

Modules can be declared as a nested module (i.e., with a STATIC modifier) or inner module, which required the separation of ModuleRoot and Module. In the case of such modules, the "ModuleRoot" would otherwise not be the root node of the tree anymore. The current design makes the module reusable.

While the return type for Unit and Property is straightforward, it might not be obvious for the Module type. The module type returns the Module reference, which can access all statically available members.

4.6 Unit

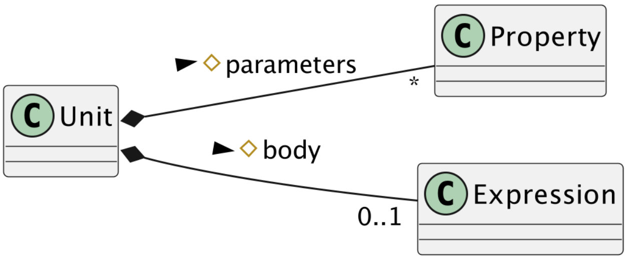


FIGURE 9: LAMP Metamodel - Unit

The Unit element is the abstract representation of a method declaration in Java, C#, and Scala, and a function declaration in Kotlin. Unit is another term in the terminology used by the Maintainability Model [3].

The signature of a unit is composed of the id and optional parameters. Parameters are modelled as Property types. Like a Unit, a Property is a declarable type and thus can contain modifiers. Although parameters cannot have access modifiers, they can have non-access modifiers (e.g., `fun run(final obj: Object)`). Parameters can also contain a default value which is captured with the "initializer". Parameters should be seen as "local properties", because they are only accessible within the scope of the declared unit.

Besides the standard unit, there are two constructs in multi-paradigm languages that require special attention when modelling them within the LAMP metamodel. The first construct is the constructor. We see a constructor as the unit that generates an instance of a module. Therefore, the constructor is modelled as a unit with the id `"ModuleId"` and the corresponding return type is the module. The second construct is the initializer, either non-static or static. It is modelled, similar to a constructor, as a unit with an empty id because it will never be called from the source code. Static initializers must include a STATIC modifier within the unit declaration modifiers.

A unit can also contain a body that describes the sequence of elements that is executed on a unit call. This body is optional due to abstract Unit types and interface Unit types not having a method body (e.g., `abstract fun doStuff();`).

Finally, a unit can contain a return type, which is always indicated within the language-specific signature of the unit. To establish a full call graph when accessing one or more declarables, this return type must be known.

4.7 Property

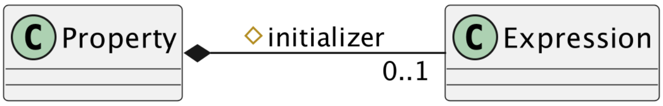


Figure 10: LAMP Metamodel - Property

Property is the last of the three declarable types within the metamodel. Property should not be confused with the C# and Kotlin property constructs. In those languages, properties contain getter and/or setter functionality. Transformations in the metamodel should convert the getter and setter functionality to units. Within the metamodel, a property contains an optional reference to a value. It is the only element within the metamodel that represents the mutable state. Therefore, when we want to detect whether a state is mutated, we must focus on operations that mutate properties.

4.8 Expression

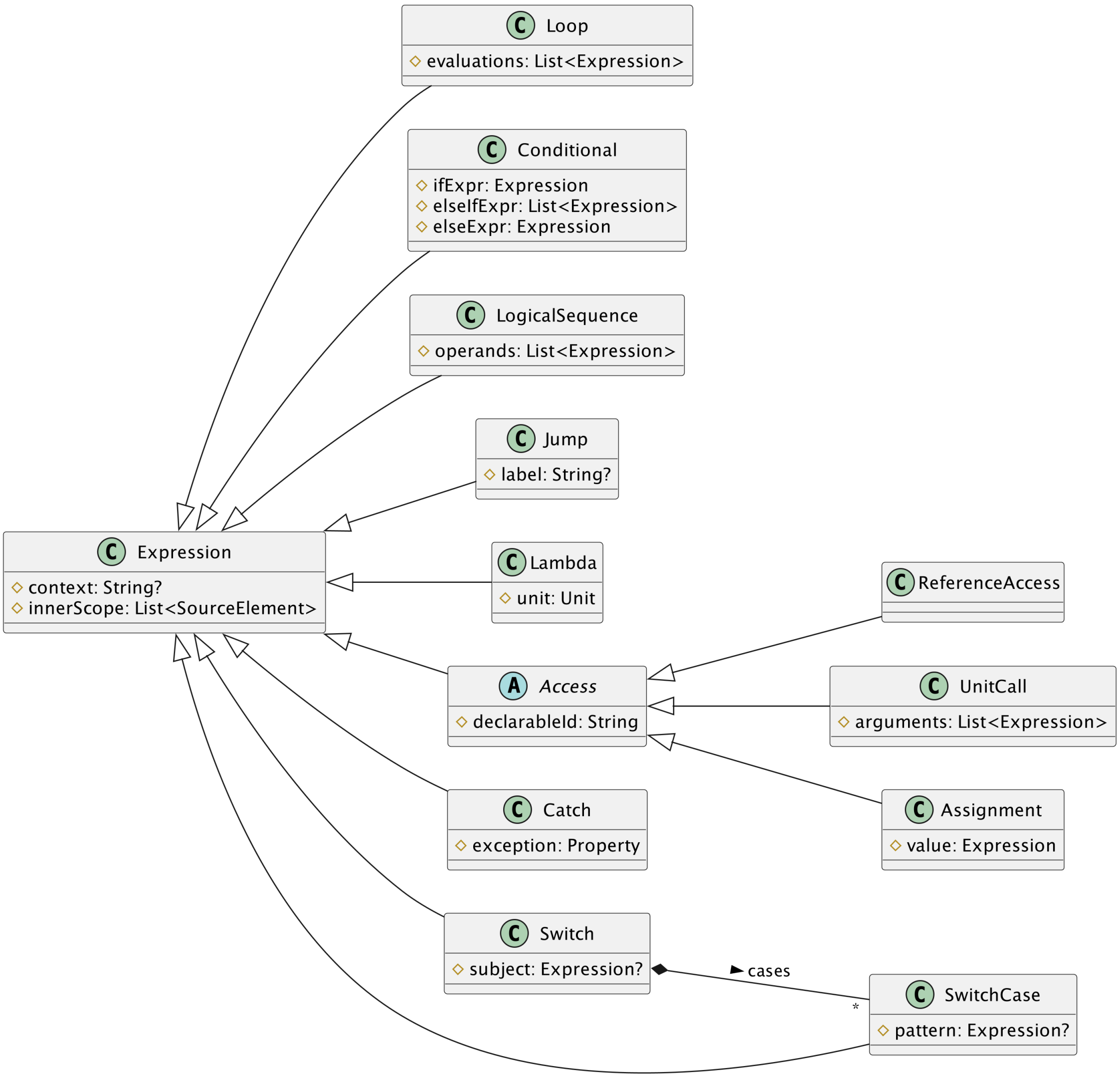


FIGURE 11: LAMP Metamodel - Expressions

Modern languages have used statements as logic blocks without a return value and expressions as logic blocks with a return value to model their languages. Although statements and expressions are different in Java, Scala took a broader approach by calling everything an expression. Java is the most limited because it has a lot of rules about how statements and expressions should be written. Scala is the least restrictive on how source code can be written and allows for high expressiveness with many paths of freedom.

Finding the middle ground in the range of restrictiveness and expressiveness can be done the Java way, by defining everything beforehand within a metamodel. This leads to a larger metamodel, which might be harder to understand. On the other hand, from a Scala point of view, a lot of freedom is given to do what you want. If this freedom is translated into the metamodel, it allows for much freedom, as in Scala variant.

Restrictiveness is only important for providing an initial structure, the metamodel, when measuring code quality. The Java compiler enforces restrictions on Java source code, which allows the metamodel to be less restricted, knowing that the transformed Java code is semantically correct. Therefore, we view statements as expressions to be able to bundle them together in the metamodel. While expressions return a value, statements will not. These return values are not important, as they can be inferred from the return type of a declarable.

Expressions can represent any type of statement or expression, although not every statement or expression is explicitly defined within the metamodel. We do not require explicit semantic information for each language construct within our metamodel to be able to compute the selected metrics.

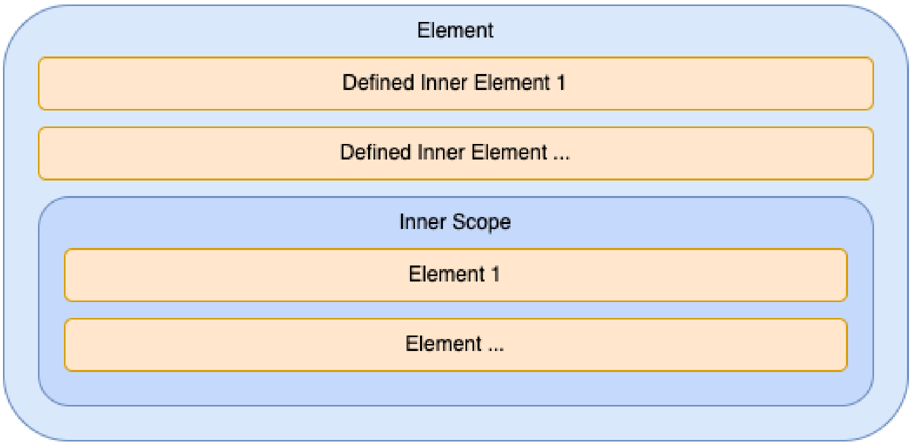


FIGURE 12: LAMP Metamodel - Structure

The most important property of Expression, and of this entire metamodel, is the idea of inner scope. As shown in Figure 11, an expression contains an inner scope, which is a list containing SourceElement types. An expression type can be seen as a scoping mechanism within the metamodel. It describes itself, namely the statement or expression, and everything that is part of its scope. In Figure 12, we show how elements can be placed within the inner scope of an element.

As shown in Listing 1, two types of units are defined. The first unit contains a simple binary expression that has two expressions in its inner scope. The first expression is PropertyAccess to access the parameter defined in the unit parameters. The second expression is an expression that we do not find interesting for our metamodel. The other unit declared in the listing follows the same modelling principle.

```
// Unit body: Expression(context=BinaryExpression) with in its inner
//       scope a PropertyAccess and an Expression (context=Literal).
fun timesThree(x: Int): Int = x * 3


// Unit body: Expression(context=Block) with innerscope Assignment & UnitCall
fun doStuff() {

  x = x * 3 // Assignment of x, value is Expression(context=BinaryExpression) with
            // in its inner scope a PropertyAccess and an Expression (context=Literal).

  println("$x is now tripled!") // UnitCall with a literal parameter
}
```

LISTING 1: Expression inner scopes

When performing an analysis using the LAMP metamodel, it can be useful to know the underlying language context corresponding to the expression. This context could be used for debugging purposes, or to allow for language-specific analysis using the language-agnostic metamodel. The "context" property is part of Expression to be able to distinguish between statements and expressions that were transformed to the language-agnostic metamodel. For each element of the source code that is being transformed, the semantic context of the expression is embedded in this property. This context is prefixed by the language from which it is transformed (e.g., "java:" or "kotlin:").

4.8.1 Loop

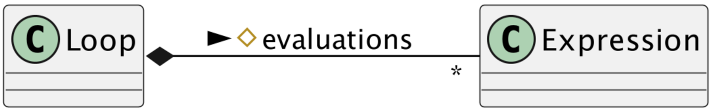


FIGURE 13: LAMP Metamodel - Loop

In the selected languages, several types of loop statements are present. We can distinguish between "for", "foreach", "while", and "do while". These languages share the common trait of looping over the scope of the statement until a certain condition is met. The condition is part of the signature of the loop statement. Therefore, this signature condition is kept separate from the inner scope of the Loop in the metamodel, modelled as the "evaluations" property.

A list is chosen because of the possibility of having multiple expressions within the "for" header. It can be a property declaration, a condition, or an updater of the property. These expressions have been bundled to make querying this signature easier. In the inner scope of Loop, the body of the loop is located.

4.8.2 Conditional

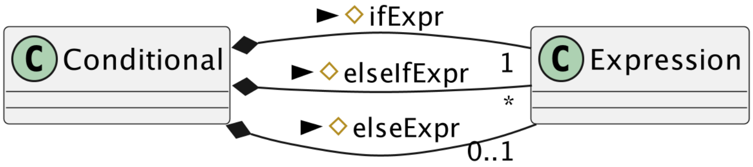


FIGURE 14: LAMP Metamodel - Conditional

There are three types of Conditional: if expressions, ternary operators, and null coalescence operators. Ternary and null coalescence operators can be viewed as syntactic sugar for the if expression, thus the properties "if", "elseIf", and "else" were chosen. The if, elseIf, and else branches of a conditional are separated because of its importance in the CC and COCO metrics. These metrics require us to explicitly model branching points to measure complexity.

For all types of conditional, there is a primary condition; if this condition is true, the inner scope of the if expression is executed. If the condition is false, we execute the elseIf expression if it is present or the else expression if it is present.

4.8.3 LogicalSequence

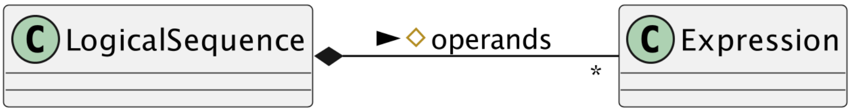


FIGURE 15: LAMP Metamodel - LogicalSequence

LogicalSequence contains the "operands" property. These operands are expressions that are placed consecutively within the source code and of which the operator between each operand is of the same type (i.e. `||` or `&&`). CC counts each operator as a branching point, increasing the complexity by one. COCO views the sequence as an increment of one in cognitive complexity.

This element is explicitly modelled as a sequence to make it easier to determine the cognitive complexity of a Unit. An alternative choice could have been to model the BinaryAnd and BinaryOr expressions separately. It was a matter of preference to choose LogicalSequence.

4.8.4 Jump

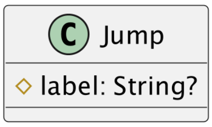


FIGURE 16: LAMP Metamodel - Jump

As the name suggests, Jump expressions describe a jump within the call graph of a program. Jumps can be a break, continue, or goto. Optionally, these jump expressions contain a label that indicates where to jump to or where to jump out of. Therefore, an optional label property is added to Jump. This element is explicitly modelled to indicate a branching point, which is required for computing CC and COCO.

4.8.5 Lambda

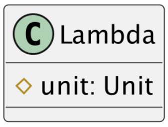


FIGURE 17: LAMP Metamodel - Lambda

Lambda is the centre piece within the multi-paradigm setting. Lambda can only be declared as a lambda expression (i.e. a parameter of a Unit or argument of a UnitCall) or as the value of a property declaration (i.e. a function literal). Lambda is modelled as a wrapper for Unit to allow for easier querying of Lambda declarations.

A lambda can contain parameters, but these are contained within the "unit" property. In some languages, the type of lambda parameter is left out because the compiler is smart enough to infer the type from the surrounding code. In our metamodel, we do not have access to these smart compiler techniques. The type information of declarable types can still be inferred from the metamodel, given that there is no type inference from external dependencies.

Like normal units, lambdas can also be called via unit calls. This is applicable to function literals for which a reference must be created to the wrapped unit. This allows unit calls to be linked to the Lambda unit. To enable the link, the id of the property is copied to the id of the Lambda unit. This enables queries to Unit types with the id of the property.

The return type of a lambda requires a separate explanation from other Declarable types. The Lambda acts as a Property value, but can be called like a Unit. Therefore, when resolving the type of the Lambda when it is called, we should return the type of the value returned by the wrapped Unit.

4.8.6 Access

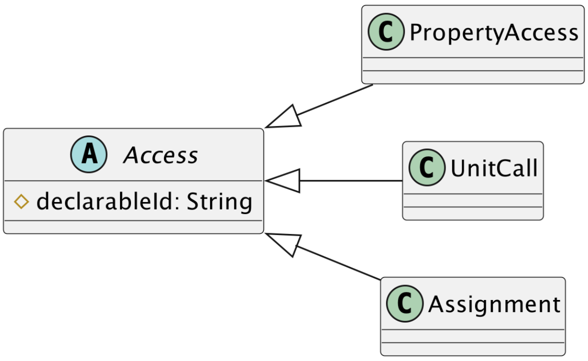


FIGURE 18: LAMP Metamodel - Access

We need to model declarable access and any changes that have been made to those declarables to be able to establish a call graph. Access is the element that holds the reference to a declarable, which we can look up by its id. Declarable can be used to perform a variety of operations. Firstly, a unit can be accessed via a call, possibly with arguments. Secondly, a property can be accessed through a reference to read the value of the property. Third, a new value can be assigned to a property.

The abstract Access element was designed because the behaviour of all these different operations is the same on a call-graph level. There is a mapping from an access operation and a declarable. The specific behaviour of each Access subtype can be implemented as an extension of the default Access element.

4.8.7 UnitCall

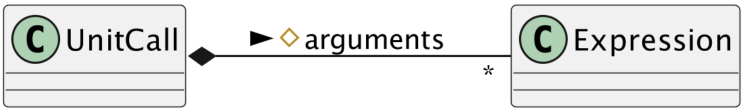


FIGURE 19: LAMP Metamodel - UnitCall

UnitCall represents a call to a unit. Like any Access subtype, it contains a reference to the corresponding declarable, which is always a Unit for unit calls. It is the element for calls to Unit and Lambda. As explained, in the case of Lambda, the wrapped unit contains the id of the property declaration.

Units can be overloaded, which means that there can be multiple units with the same id. So, when we check to see which unit is being referred to, we must also look at the type of each argument and see if it matches the type of the Unit parameter.

4.8.8 ReferenceAccess

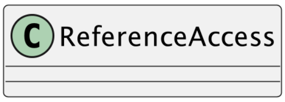


Figure 20: LAMP Metamodel - ReferenceAccess

ReferenceAccess models the mapping from a place in the source code to one of the Declarable types via a reference id. We modelled ReferenceAccess as a subtype of Access to better distinguish between the different types of Access during metamodel traversal.

As said above, all declarable types can be referenced. For a module, this means that we reference the module, which allows access to all static declared members of that module. For properties, this means getting the value that corresponds to the property. For units, this means creating a method reference.

4.8.9 Assignment

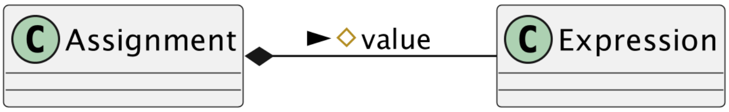


Figure 21: LAMP Metamodel - Assignment

Assignment is the element that models the assignment on a Property by reference. Assignment is also the generalisation for operator assignments and unary increments and unary decrements. These special types of assignments are part of Assignment because they also mutate state, which is all the information required for our metrics.

Assignment types should be queried and analysed to detect mutations within the source code. This element is therefore essential for the detection of the Lambda function with Side Effects metric. If an assignment is done on a referenced property that is outside of the scope of the enclosed Lambda, the Lambda contains a side effect.

4.8.10 Catch

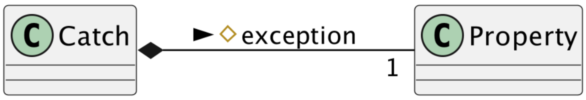


Figure 22: LAMP Metamodel - Catch

The catch statement is explicitly modelled to compute the cognitive complexity. The exception property of Catch tells us about the type of exception that is caught.

4.8.11 Switch

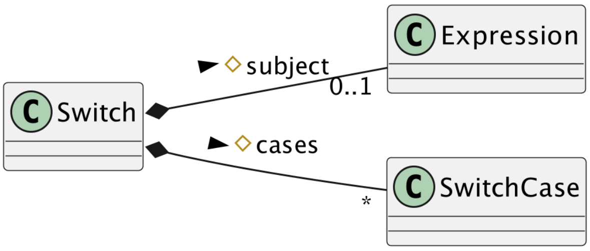


FIGURE 23: LAMP Metamodel - Switch

The Switch element is an important factor within MP programming languages due to the pattern matching features it provides. Within Java and C#, it is called a `switch` statement or expression. Its keyword in Kotlin is `when`, and it is `match` in Scala.

Within Java, C#, and Scala, Switch always contains a subject, but in Kotlin, this is optional. Therefore, this "subject" property within the metamodel is optional. A Switch also contains zero or more cases that can be matched on. These cases will be explained in more detail in the section hereafter.

4.8.12 SwitchCase

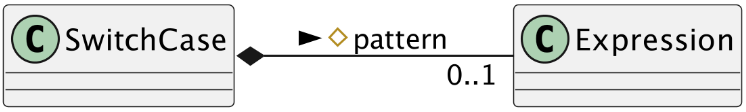


FIGURE 24: LAMP Metamodel - SwitchCase

A SwitchCase can only be found in Switch. It is modelled as an extension of Expression to enable the extension of the properties "metadata", "context", and "innerScope". A SwitchCase consists of an optional "pattern" property, which is similar to the condition part of an if statement. When no pattern is given, this switch case can be seen as the default case. When a case matches, the inner scope of that case is executed.

5 Code Quality Assurance Framework

We defined our goal, questions, and metrics in Section 3 and the metamodel was designed in Section 4. To explore the extent to which we can abstract from language-specific constructs and compute code quality metrics on a language-agnostic level, we design a framework that describes the workflow for code quality assurance. With this framework design, we are able to answer RQ3.

This framework is inspired by other studies that did not capture the properties of multi-paradigm programming languages, which this framework tries to accomplish. These other frameworks are the Maintainability Model by Heitlager et al. [3] that computes a maintainability score with the identification of root causes, the M3 model by Basten et al. [64] which is a common source code model for metaprogramming purposes, and SmellFW by Moha et al. which can detect code smells and code metrics [63].

This framework is useful for creating uniformity in a code quality assurance process involving multiple programming languages and paradigms. At the start of this research, we focused on three major phases to be encapsulated by a framework, namely transformations, evaluation, and providing suggestions. During background research on code quality metrics, we found that there was a lack of threshold data available to provide evidence-based code quality improvement suggestions. Therefore, the third major phase is not included in this framework design. The goal of the framework, as defined in Section 3.2, is to analyse and evaluate source code with respect to maintainability.

In the following subsections, we will go over the requirements of the framework, the prototype, and the design of the framework and its workflows. With these topics covered, we show the significance of designing a framework for reaching our goal defined in Section 3.2. Using the framework design, we can create the prototype implementation which is essential for answering RQ4.

5.1 Requirements

In this section, we define several requirements that our framework must meet to be able to call it a code quality assurance framework. The following requirements have been written with the RFC2119 prioritisation standard [65]. The system must be able to:

R.01 parse Java class files to one Java AST per top-level class declaration

R.02 parse C# class files to one C# AST per top-level class declaration

R.03 parse Scala class files to one Scala AST per top-level class declaration

R.04 parse Kotlin class files to one Kotlin AST per top-level class declaration

R.05 transform language-specific ASTs to the LAMP metamodels

R.06 transform LAMP metamodels to serialised LAMP files

R.07 transform serialised LAMP files to LAMP metamodels

R.08 calculate metrics using LAMP metamodels

R.09 locate referencing class file of a LAMP metamodel

R.10 write metric results to CSV file per metric category (i.e., module and unit)

R.11 give suggestions for improvement based on metric results

R.12 give a maintainability rating based on metric thresholds compared to the metric results

The system should be able to:

R.13 import source code project from a Git repository

5.2 Prototype

The prototype is developed in Kotlin. Kotlin is interoperable with Java, which is the language on which a multitude of useful libraries for this prototype depend. Kotlin provides more flexibility and requires less boilerplate code, which is a personal preference.

Several system requirements have been defined in Section 5.1. These requirements have been written to grant a perspective on the top-level structure of the framework prototype and, therefore, its major features.

Our prototype has not met every requirement. Transformations of C#, Scala, and Kotlin are currently not possible and therefore **R.02**, **R.03**, and **R.04** have not been fulfilled. Furthermore, the prototype currently does not support the suggestions phase, namely **R.11** and **R.12**.

5.2.1 Metamodel Specification

In Section 4, the metamodel design was described. To use this metamodel within the prototype, a specification for this metamodel needed to be created. This specification was written using the XML Schema Definition (XSD) language [66]. An advantage of using XSD for the metamodel specification is the large support for tooling, such as checking validity of the XML document according to its specification (i.e., XSD), generating code, and querying the conforming XML files (e.g., with XPath). A tool called JAXB[9] could be used to generate Java class files that comply with the XSD specification. For the prototype, the JAXB extension called JAXB-Visitor[10] (version: 3.0.0) was used. This library generated these Java class files with the addition of the Visitor pattern[11]. This design pattern allows for a convenient traversal of metamodel instances.

[9] `https://jakarta.ee/specifications/xml-binding/3.0`

[10] `https://github.com/massfords/jaxb-visitor`

[11] `https://refactoring.guru/design-patterns/visitor`

5.3 Framework Design

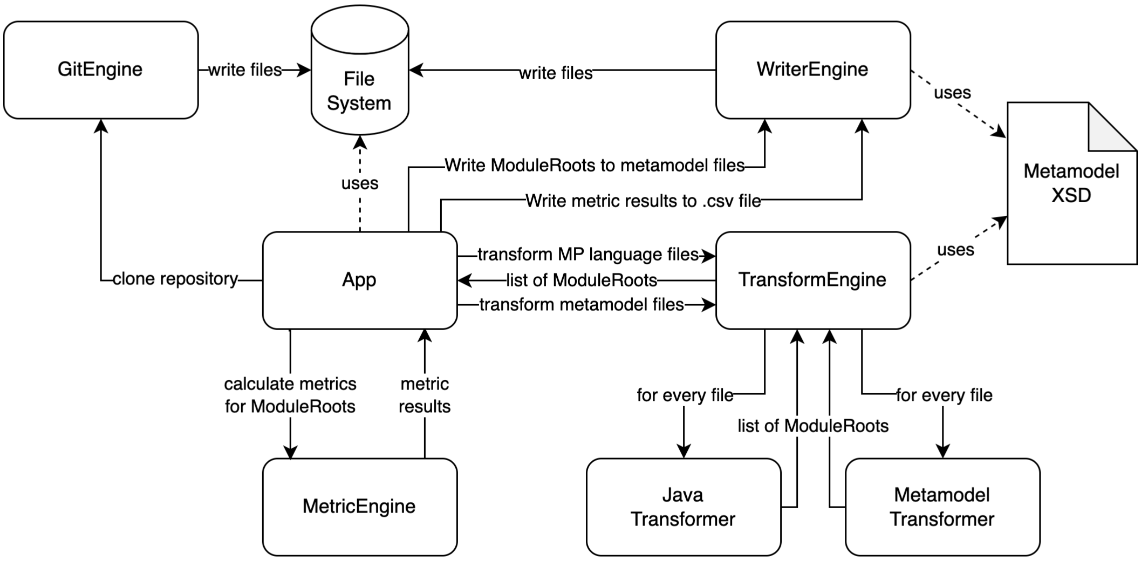


FIGURE 25: Framework Design Overview

Due to the unknowns at the start of this project, the architecture for this framework had to be designed in a modular way. The components of the framework were created iteratively without knowing the implementation details of the entire framework beforehand. The disadvantage of this approach was the possibility that changes need to be made to earlier modules if a module coming at a later stage requires extra details from previous modules.

5.3.1 App

The App is the glue for all the engines within the system. It is responsible for handling the data flows between the engines. With the app, external libraries could interface with the framework. In the prototype, App is interfacing with a Text-based User Interface (TUI) and a Command Line Interface (CLI).

5.3.2 GitEngine

GitEngine is responsible for retrieving source files from a Git repository hosted online. It takes as input the url of the repository, along with the output directory. Upon a clone request, it retrieves the files from the external version control and writes them to the output directory. If the output directory already hosts the Git repository, a Git Pull action is done that only writes changed files to the output directory.

5.3.3 TransformEngine

The TransformEngine is responsible for transforming input files, either MP language class files or metamodel XML files, to metamodel ModuleRoot instances. The TransformEngine can be extended with supported language transformers. Depending on the input file's extension, the correct transformer to parse the file can be selected.

The Metamodel Transformer is responsible for transforming `.lamp` XML files into metamodel class instances. `.lamp` is the custom file extension that is used for metamodel files that have been written to the file system. Due to the choice in tooling within the prototype, transforming the XML files into class instances required little code. Using the JAXB Unmarshaller, XML files conforming to our XSD were transformed into metamodel instances.

The Java Transformer is created using the ANTLR4 tool described in Section 2.4.7. The Java grammar provided by the ANTLR grammar-v4 repository was used to generate a Java Lexer and Parser (commit hash: 55f74807). The generated visitor was extended by the Java transformer to implement specific rules per semantic element within the Java source code.

5.3.4 MetricEngine

The MetricEngine is responsible for computing metric results. There are three types of metrics (i.e., unit, module, and coupling) and two types of metric categories (i.e., module and unit). Module- and unit-level metrics can be calculated in parallel because they are independent. Coupling metrics require operations with more complexity.

5.3.5 WriterEngine

The responsibility of WriterEngine is writing output files to the file system. There are two types of outputs, namely metric results and ModuleRoot instances. When metric results are collected, they are categorised by module or by unit. The WriterEngine will output each category in a separate CSV file to the designated output directory. When a language has been parsed into ModuleRoot instances, they need to be saved to the file system. The WriterEngine will write each ModuleRoot to a separate output file, using the `.lamp` extension, within the designated output directory.

5.3.6 Framework Pipeline

The framework can be seen as a pipeline, with a project of source code as input in one of our selected languages and as the final output of the metric results. To give a practical example of application processes, we present Figure 26 where Java source code is transformed into an LAMP metamodel instance and written to LAMP XML files for later use.

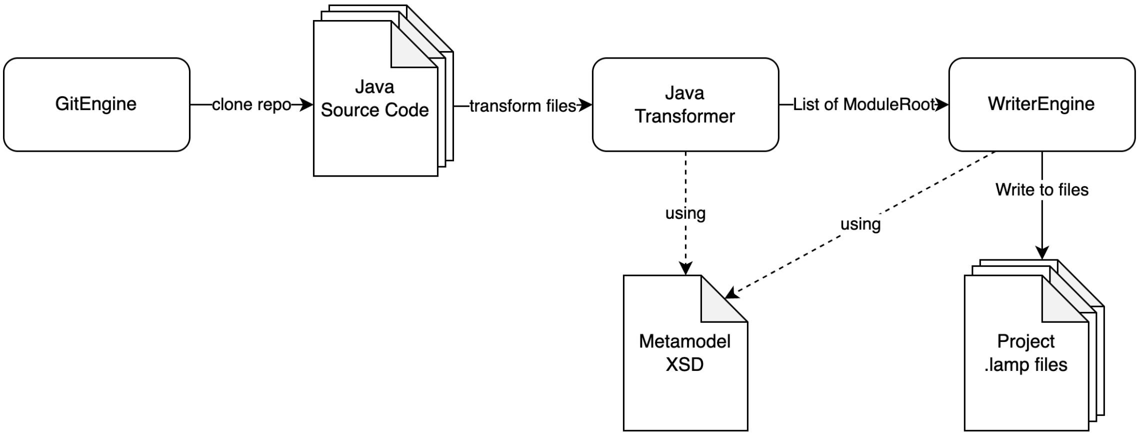


Figure 26: Framework Language Transformation Workflow

At a later stage, as shown in Figure 27, the saved LAMP XML files are used to compute metrics. The metric results are written into output files for analysis.

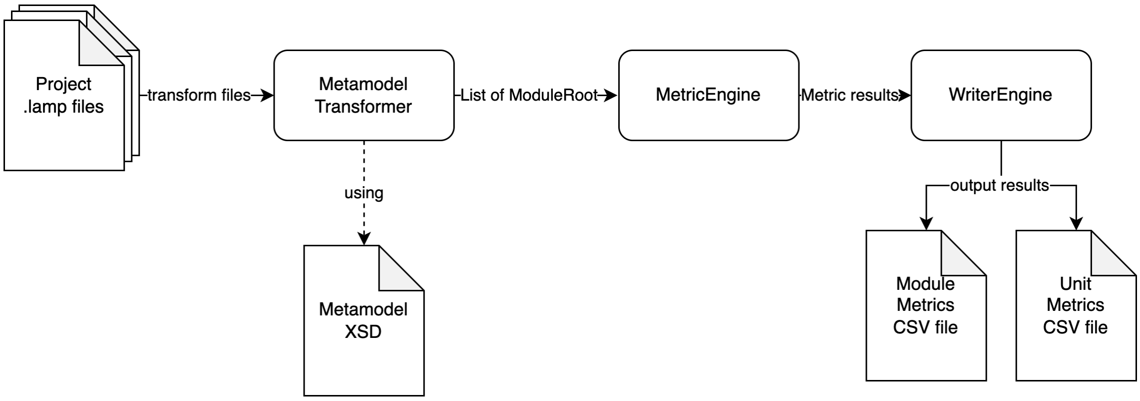


FIGURE 27: Framework Evaluation Workflow

5.4 Metric Categories

The MetricsEngine is responsible for computing all metrics that were covered in Section 2.5. With the designed metamodel, we must be able to compute these metrics. The metrics are grouped into three categories: unit metrics, module metrics, and coupling metrics.

Module metrics are grouped per module, where the module name is its unique name within the project (i.e., `"componentId.moduleId"`). Unit metrics are grouped per unit, where the unit name is its unique name within the project (i.e., `"componentId.moduleId://unitId"`). We separate component hierarchy from units due to our naming convention for constructor calls. This unique naming convention makes it easier to distinguish between Modules and Units. Coupling metrics are more complex to compute using the representation of the metamodel. Coupling metrics are grouped per module in the same way as the module metrics. Some metrics require multiple or even all modules to be accessible at the same time to be computed. These metrics require resolving references to other parts of the project through type resolution. For type resolution purposes, a symbol tree was created which stores information about module inheritance and maps Access references to Declarable types.

6 Evaluation

This section aims to provide a comprehensive evaluation of the correctness and accuracy of the framework. The chapter is divided into three subsections, each focusing on a specific aspect of the evaluation process. The first subsection evaluates the computation of metrics using the metamodel in detail, taking into account the limitations of the metamodel and its effect on computing the desired metrics. The second subsection goes on to evaluate the limitations and solution paths of the framework in more detail. The third subsection uses the framework prototype and two benchmarks on five open-source Java projects to evaluate the prototype's ability to accurately compute code quality metrics in real-world codebases. At the beginning of each subsection, a summary of findings is given to provide a brief description of what was found.

6.1 Metric Evaluation

In this section, we will evaluate how each metric is computed using the metamodel. We analyse how each metric computation requirement is mapped onto metamodel elements and the metric limitations relating the current metamodel design. In this evaluation, a satisfying result is reached when all requirements for each metric can be mapped onto its language-agnostic counterparts.

Summary of Findings: We found for every selected metric a mapping from the requirements for computing the metric to the language-agnostic representation. Some metrics require us to traverse multiple ModuleRoots to capture inheritance information or references to source elements outside of one ModuleRoot. This information can be extracted with some workarounds.

6.1.1 Cyclomatic Complexity (CC)

Cyclomatic Complexity was originally computed by counting the number of branch points within a unit. Using the metamodel, the creators of these branch points had to be identified. For every top-level declared unit present in a module, a CC count is performed. The count starts by default at one. When traversing the body of the unit, the count is increased by one for some of the elements that are found during traversal. These elements are as follows:

- Unit: increase by one.
- Catch: increase by one.
- Conditional: increase by one for every if, elseIf and else entry.
- SwitchCase: increase by one if there is no pattern entry (it is the default switch case).
- Loop: increase by one.
- LogicalSequence: increase by the count of operators present in the logical sequence, retrieved by measuring the size of the "operands" property minus one.

In this implementation, the choice was made to only consider the count of CC for units that are a top-level member of a module. Nested units and lambdas defined within the inner scope of the outer unit are considered to be part of the unit. It can be argued that these nested units and lambdas actually reduce the complexity within a unit because they introduce descriptions of procedures.

6.1.2 Cognitive Complexity (COCO)

The Cognitive Complexity definition given in Section 2.5.3 described how COCO is computed. Similarly to CC, COCO is measured on only top-level member units of modules. For COCO, there are two counters active in the computation, namely the nesting count and the total count. The nesting count increases for some element types when traversing the inner scope, but the nesting level decreases when leaving that inner scope. This behaviour applies to:

- Conditional: increase for every inner scope traversal of an if, elseIf and else.
- Switch: increase before traversing the switch cases but after traversing the subject.
- Loop: increase while traversing the inner scope.
- Catch: increase while traversing the inner scope.
- Unit: increase while traversing the unit body. This also covers the lambda case from the COCO definition, due to Lambda being a wrapper for Unit.

The following elements increment the total COCO count for each encounter in the traversal:

- Conditional: increase by the current nesting level. Additionally, increase by one for every if, elseIf and else.
- Switch: increase by the current nesting level plus one.
- Loop: increase by the current nesting level plus one.
- Catch: increase by the current nesting level plus one.
- Jump: increase by one if the "label" property exists.
- LogicalSequence: increase by one.
- UnitCall: increase by one if it references the Unit, thus being a recursion cycle.

6.1.3 Parameter Count (PC)

The Parameter Count of a Unit is only calculated for top-level member units of modules. This metric is computed by counting the entries within the "parameters" property of Unit. This metric does not count the parameters of a Lambda unit because they do not occur as a top-level member of a module.

6.1.4 Unit Lines of Code (ULOC)

Lines of code per top-level unit of a module are computed by traversing all the source elements contained within the unit. To collect which lines contain code, a set of numbers is initialised. For every source element that is part of the unit, the startLine and endLine of the source element are added to the set of numbers. Only new lines will be added to the set of line numbers. Therefore, the total ULOC can be computed by retrieving the size of the set.

6.1.5 Lambda Lines of Code (LLOC)

Lambda Lines of Code is computed in the same way as ULOC, but only LOC that are part of Lambda are counted. Due to the categories in which metric computations get collected, only the LLOC within the body of a Unit are counted. A Lambda that is located in the "initializer" of Property is ignored. This is a current limitation of the framework that could be solved by introducing a property metric category.

6.1.6 Module Lines of Code (MLOC)

The module lines of code can be retrieved in a way similar to that previously described in ULOC and LLOC. In this instance, the startLine and endLine of every source element within the module is added to the set of line numbers. After traversing the entire module, the size of the set is computed as MLOC.

6.1.7 Length of Message Chain (LMC)

The LMC metric is computed by counting the length of each message chain within a unit and retrieving the maximum of those lengths. The message chain can be detected by traversing a UnitCall. When the unit call contains only one element within its inner scope, and that element is a unit call as well, there is a chain of units. When the same traversal is repeated for consecutive unit calls, the total length of that message chain is found.

6.1.8 Depth of Looping (DOL)

The DOL metric is computed by counting the maximum depth of looping within a unit. Similarly to the LMC metric, it recursively checks if a Loop contains another Loop element within its inner scope. The depth is increased for each nested loop, and the maximum depth is computed after the entire unit has been traversed.

6.1.9 Weighted Method per Class (WMC)

The WMC metric is computed by summing the CC computations for each top-level member of type Unit within the module.

6.1.10 Cognitively Weighted Method per Class (CWMC)

The CWMC metric is computed by summing the COCO computations for each top-level member of type Unit within the module.

6.1.11 Number of Units (NOU)

The NOU metric is computed as the size of all top-level members of the Unit type within the module.

6.1.12 Lambda Count (LC)

The LC metric is computed as the size of all source elements of the Lambda type contained within a module.

6.1.13 Inheritance Tree

Due to the framework only capturing one module within the metamodel, we must create inheritance trees to capture inheritance relationships between modules. The inheritance trees are created by iterating through all modules, where each Module becomes a node. When a module is an extension of another module, an edge is created between the two module nodes, describing the parent-child relationship. After iterating through all nodes, a list of inheritance trees is established.

Our framework currently only transforms the source code located within a software project, thus excluding external dependencies. Therefore, we create a mock node for each extended module that was imported from an external dependency. This means that we cannot look further up the inheritance tree than the external module referred to.

6.1.14 Depth of Inheritance Tree (DIT)

The DIT metric can be computed as shown in Algorithm 1 with the Inheritance Model. This metric counts the number of edges between the evaluated module node and the root-level parent node.

```
Data: $inheritanceTrees$, $curModule$ to evaluate
Result: Computed $DIT$ metric for evaluated module
$count \leftarrow 0$;
$currentNode \leftarrow inheritanceTrees.findNode(curModule)$;
while $currentNode.parent \neq null$ do
    $count \leftarrow count + 1$;
    $currentNode \leftarrow currentNode.parent$;
end
```

Algorithm 1: Compute DIT for a module

6.1.15 Number of Children (NOC)

The NOC metric can be computed as shown in Algorithm 2 with the Inheritance Model. This metric counts the number of child edges of the evaluated module node.

```
Data: $inheritanceTrees$, $curModule$ to evaluate
Result: Computed $NOC$ metric for evaluated module
$node \leftarrow inheritanceTrees.findNode(curModule)$;
$NOC \leftarrow node.children.size$;
```

Algorithm 2: Compute NOC for a module

6.1.16 Symbol Tree

The Symbol Tree is an extension of the Inheritance Tree. Where the inheritance tree is responsible for capturing the inheritance relationships between modules, the symbol tree is responsible for capturing the relationship between Access types and Declarable types. The symbol tree can be used to resolve the return type of an Access element. This tree is essential to obtain type information using only the metamodel design. The symbol tree uses the inheritance tree to resolve inherited declarables. The tree is also used to establish a connection from the import clauses of a module to the metamodel representation of the imported module. The symbol tree uses the metamodel tree structure extended with helper methods to find declarables within the scope of an expression, unit, property, or module.

6.1.17 Coupling Between Objects (CBO)

The CBO metric can be computed as shown in Algorithm 3 using the Symbol Tree. For this metric, all modules within the project must be traversed to establish all possible couplings for each module.

```
Data: $symbolTree$, $curModule$ to evaluate
Result: Computed $CBO$ metric for evaluated module
$moduleCoupling \leftarrow Map(Module, Set(Module))$;
for $module \in modules$ do
    $refs \leftarrow symbolTree.findMembers(module = curModule, type = Access)$;
    for $access \in refs$ do
        $declarable \leftarrow symbolTree.findDeclarable(ref = access)$;
        $otherModule \leftarrow declarable.getModule()$;
        if $otherModule \neq module$ then
            $coupledModules[module].add(otherModule)$;
            $coupledModules[otherModule].add(module)$;
        end
    end
end
$CBO \leftarrow coupledModules[curModule].size$;
```

Algorithm 3: Compute CBO for a module

6.1.18 Response For a Class (RFC)

The RFC metric can be computed as shown in Algorithm 4 using the Symbol Tree. We measure the response set spanning all Units, excluding constructors and initializer units.

```
Data: $symbolTree$, $curModule$ to evaluate
Result: Computed $RFC$ metric for evaluated module
$responseSet \leftarrow Set(DeclarableId)$;
$units \leftarrow symbolTree.findMembers(module = curModule, type = Unit)$;
$units \leftarrow units.filter(id == curModule.id || ””)$;
for $unit \in units$ do
    $responseSet.add(unit.id)$;
    $calls \leftarrow symbolTree.findElements(scope = unit.body, type = UnitCall)$;
    for $call \in calls$ do
        $responseSet.add(call.declarableId)$;
    end
end
$RFC \leftarrow responseSet.size$;
```

Algorithm 4: Compute RFC for a module

6.1.19 Lack of Cohesion in Methods (LCOM)

The LCOM metric can be computed as shown in Algorithm 5 using the Symbol Tree. The `symbolTree.getUnitPairs()` unit collects every combination of two units possible.

```
Data: $symbolTree$, $curModule$ to evaluate
Result: Computed $LCOM$ metric for evaluated module
$unitPairs \leftarrow symbolTree.getUnitPairs(of = curModule)$;
$cohesionCount \leftarrow 0$;
for $(unit1, unit2) \in unitPairs$ do
    $i \leftarrow symbolTree.findElements(scope = unit1.body, type = ReferenceAccess).toSet()$;
    $j \leftarrow symbolTree.findElements(scope = unit2.body, type = ReferenceAccess).toSet()$;
    if $i.intersect(j).size > 0$ then
        $cohesionCount \leftarrow cohesionCount + 1$;
    end
end
$LCOM \leftarrow (unitPairs.size - cohesionCount) - cohesionCount$;
```

Algorithm 5: Compute LCOM for a module

6.1.20 Lambda Functions w/ Side Effects (LSE)

This LSE metric measures the amount of lambda functions that have side effects. First, the Lambda unit is evaluated for Assignment elements. When Assignment elements are present that mutate a Property outside of the lambda, it can be marked as having side effects. Since lambdas can also contain unit calls, we must be cautious of impure units that are being called. Units can also call units themselves, thus we must traverse the call graph to efficiently detect mutations.

6.2 Framework Evaluation

We have analysed the constructs of the major MP programming languages and compared them with each other. We have developed a metamodel that can compute metrics based on an abstraction of these constructs. In the previous subsection, we evaluated how well these metrics can be computed on a language-agnostic level. In that evaluation, several limitations became noticeable, which we cover in this section. In addition to covering limitations, we will also cover implementation-related findings of the framework prototype.

Summary of Findings: We must be cautious of the procedures that are defined for type resolution on a language-agnostic level. While type resolution in one programming language is complex but clearly defined, we must know the behaviour of constructs of all languages individually on a language-agnostic level. This requires us to model behaviour of each language explicitly to contain behavioural information in a language-agnostic representation. Therefore, transformation conventions from individual languages to a language-agnostic representation must be described in future work to produce uniform results in the evaluation phase of the framework.

6.2.1 Type resolution

First and foremost, a concept that was mentioned throughout this thesis was type resolution. Referencing the correct declarable can be difficult because the referenced declarable can be located in more than one scope. The search for the declaration that matches the identifier will start from the current scope and move outward. Type resolution is required when dealing with metamodel Access subtypes. This Access contains an id which it references. There is a certain order in which this id can be found within the framework. The following steps only scratch the surface of the resolution.

1. Search in the current scope for an element of type Declarable matching the id.
2. If the current scope has a parent scope, repeat step 1 for the parent scope. Otherwise, continue to step 3.
3. If the current module is an extension of another module, search for the Declarable id in the member list of the parent module, considering only the accessibility modifiers `PROTECTED` and `PUBLIC`. Repeat this step until there is no parent to evaluate.
4. Search for the Declarable id in each imported module's member list, considering only declarables with the `PUBLIC` accessibility modifier.
5. Search for the Declarable id in the other modules of the same component, considering only declarables with the `PUBLIC` accessibility modifier.

There can be a multitude of ways in which a declarable is referenced. The following is a potentially incomplete list of ways to refer to a declarable.

- Reference by name: The variable is referenced as a simple identifier, e.g., `x`. We must look from the current scope outward for the correspondingly defined declarable. The declarable must be present within the same module.
- Reference by return value: The variable is referenced by the return value of a declarable, e.g. `getVar().x` or `objectVar.x`. In this case, we must know the return value of the prefix to know where to find the declarable.

- Reference by module: The variable being referenced is a static declarable of a module, e.g. `AClass.X`. The name of the module is used to signal in which module we can find the declarable. In this case, we must search for AClass within the scope of this component or one of the imports within this ModuleRoot.

- Reference by full name: This type of reference is quite similar to the previous "reference by module", but now we are not using the internal component structure to find the module. When referencing by full name, e.g. `nl.utwente.AClass.X`, it is fully specified where to look for the declarable.

- Reference by keyword: Languages also have some reserved keywords for special types of references. These reference keywords indicate where to find the reference. There are two of those cases: references to the same instance (e.g., `this` or `self`) or references to the instance within the scope of the parent module (e.g., `super`).

- Reference through polymorphism: When an Access return value is cast to another type, that type is the type to look for the reference instead of the type of the Access return value.

- Reference with type inference: There are situations in the selected languages where the type of reference is inferred through the scope of the referent.

6.2.2 Transformation Conventions

The designed metamodel allows some freedom to model elements from a language-specific representation to a language-agnostic representation. This choice allowed us to reuse elements within the metamodel to keep the entire metamodel relatively concise. Having this freedom is not a problem, but it requires the developer of the language transformations to be cautious about what every construct is transformed into. From a language-agnostic metric computation perspective, language-agnostic representations of a specific construct written in Java or C# must contain the same kind of elements. This calls for transformation conventions, including naming conventions. Some examples include: how to model anonymous classes, how to model arrays, and naming conventions for property getters/setters.

The second important aspect to touch upon is the modelling of source code behaviour. It is yet unclear to what extent we must transform the internal behaviour of a programming language into a language-agnostic representation. For example, it is known that we must transform declarables in the Kotlin source code by adding the `FINAL` modifier because all declarables are effectively final, unless they contain the `open` modifier. We do not have a complete view of all the behavioural properties of the selected languages.

6.2.3 Pattern Matching

The concept of pattern matching, which comes from the functional paradigm, is not implemented within our metamodel. The reason for its absence is due to the lack of literature that we could find on pattern matching relating to code quality. Therefore, the pattern matching types, as detailed in Table 2, have not been explicitly modelled as patterns within SwitchCase elements.

6.2.4 Annotations

Annotations do not change the state of the source code, but can be annotated with additional metadata. It is often used by metaprogramming tools to insert or transform the source code to include new types of functionality, e.g. the annotations used by the Spring framework. Annotation generated code can be queried at run-time using a method called reflection.

6.3 Benchmark Evaluation

To answer RQ4, we want to compare the computations with our language-agnostic framework with language-specific code quality tools. With the benchmark data from the language-specific tools, we can assess the correctness of our framework. In this section we will explain the evaluation method, list our benchmarks & analysed projects, outline the evaluation results for each project, and finally compare these results.

Summary of Findings: For each project, we compared the results from our framework prototype with the benchmark results. This comparison shows that our framework can, for the most part, correctly compute the code quality metrics on a language-agnostic level. There are a few constructs that were not captured by our framework due to implementation choice differences between the framework and the benchmarks. Aside from these implementation choice differences, we can conclude that the framework can compute metrics very accurately compared with language-specific tools. We found that we can capture all the required constructs to calculate the metrics while measuring it on a language-agnostic level. As a result, the framework can be considered more versatile than language-specific tools because of its support for multiple languages.

6.3.1 Evaluation method

In this section, we explain how and why we evaluate our prototype using benchmarks. Based on this evaluation, we can provide a verdict on the correctness of our framework.

For every analysed project, we calculate the percentage of difference between the prototype metric result and the benchmark metric result. This yields the difference distribution of each metric per project as shown in Figures 28, 29, 30, 31, and 32. Furthermore, we show the percentage of exact matches in the results for each metric per project in Tables 9, 11, 13, 15, and 17. By analysing the difference distributions, we can show the extent to which our framework is able to compute code quality metrics on a language-agnostic level.

To know how we should interpret the differences, we must know what satisfactory results are. A satisfactory result means that the majority of prototype metric results match the benchmark metric results. We are aware of possible implementation differences, and therefore we are also satisfied with near-matching results as long as the difference can be attributed to such implementation choices. Because we express the results of each project in percentages during this evaluation, we provide per project (Table 10, 12, 14, 16, and 18) a table that expresses the maximum difference in absolute numbers at the 90th, 95th, 97.5th, and 99th percentile.

To consistently and efficiently evaluate why differences occur between our prototype and a benchmark tool, we analyse the 95th percentile outliers for each benchmarked metric in each analysed project. By using the 95th percentile outliers, we have a sufficient set of elements that show relatively large differences, while the set stays relatively small enough to analyse each reason behind an outlier by hand. The reason behind the occurrence of an outlier can be categorised as an implementation choice of our framework or the benchmark, or a bug in the prototype or benchmark. We summarise the reasons behind the differences in evaluated metrics in each project in Section 6.3.9.

6.3.2 Benchmarks

The goal of our evaluation is to be able to compare all metrics that were used within our framework with the benchmark data. Only transformations from Java code are currently possible with the framework, which requires us to use Java code quality tools for benchmark purposes. Due to a lack of available tools that could process Java 17 code, we were only able to retrieve benchmark data with two tools.

Designite [56] for Java is used to compute metrics for Java projects. It can compute metrics at the class level, namely Lines of Code (LOC), Number of Methods (NOM), Weighted Methods per Class (WMC), Number of Children (NC), and Depth of Inheritance Tree (DIT). It can also compute metrics at the method level, namely, Lines of Code (LOC), Cyclomatic Complexity (CC), and Parameter Count (PC). SonarQube Community Edition is used to compute the Cognitive Complexity (COCO) metric on the module level for Java projects. In Table 6, we list the language-agnostic metrics and their respective benchmark metric counterparts from which we can retrieve the benchmark data.

LAMP Metric		**Designite**	**SonarQube**
MLOC	Module Lines of Code	LOC	
ULOC	Unit Lines of Code	LOC	
CC	Cyclomatic Complexity	CC	
WMC	Weighted Method per Class	WMC	
CWMC	Cognitively WMC		COCO
DIT	Depth of Inheritance Tree	DIT	
NOC	Number of Children	NC	
PC	Parameter Count	PC	
NOU	Number of Units	NOM	

Table 6: Metric Benchmarks

6.3.3 Analysed Projects

To analyse the performance of our metamodel, a search was carried out for mature open-source projects. We use SEART [67] to find open source software projects on GitHub. Due to our framework currently only supporting Java transformations, we were restricted to Java projects in our search. To select mature and up-to-date Java projects, the search query was enhanced with a minimum of 1000 commits, a minimum of 10000 stars, and the last commit after 1 September 2022. To limit the size of files that must be analysed, we chose only to analyse one source code folder (i.e., compilable folder) of every selected project. The search resulted in 103 potential projects to analyse. From this set, we selected the five projects listed below. These projects are developed by different individuals, varying in size, popularity, and application.

- **RxJava**[12]: RxJava is a library for composing asynchronous and event-based programs using observable sequences for the JVM.
- **Jenkins**[13]: Jenkins is an automation tool that can host custom CI/CD pipelines.
- **Mockito**[14]: Mockito is a mocking framework for unit testing in Java. The framework was created by Szczepan Faber and is among the most popular mocking tools for Java.
- **ANTLR4**[15]: ANTLR4 is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It was created by Terence Parr and Sam Harwell.

[12] `https://github.com/ReactiveX/RxJava`
[13] `https://github.com/jenkinsci/jenkins`
[14] `https://github.com/mockito/mockito`
[15] `https://github.com/antlr/antlr4`

- **Lottie**[16]: Lottie is an engine for rendering Adobe After Effects animations natively on Android, iOS, Web, React-Native, and Windows. lottie-android is the Android variant of Lottie, written as a Java project maintained by Airbnb.

Project Statistics

The statistics about these project repositories on GitHub are listed in Table 7, sorted by LOC size. We expect possible differences in the evaluation results of these projects to be attributed to implementation choices of the project contributors.

Repository	**Git Hash**	**Commits**	**Stars**	**Source Folder**	**LOC**
rxjava	0eea996	6.0k	46.8k	src/main/java	96.9k
jenkins	f0be0ca	33.3k	20.4k	core/src/main/java/jenkins	27.8k
mockito	8b96cc1	5.9k	13.7k	src/main/java	20.9k
antlr4	1f9a474	9.1k	13.9k	tool/src	17.1k
lottie-android	4794facf	1.5k	33.5k	lottie/src/main/java	16.6k

Table 7: GitHub information of analysed projects

Project Evaluation Remarks

For the evaluation of metric results, we can only evaluate module and unit entries that exist in both LAMP metric results and benchmark metric results. In Table 8, we show for each project how many modules and units were evaluated per tool. The last row called *Combined* shows the number of modules and units that were joined by DeclarableId between LAMP, Designite and SonarQube.

	RxJava		**Jenkins**		**Mockito**		**ANTLR4**		**Lottie**	
	#M	#U	#M	#U	#M	#U	#M	#U	#M	#U
LAMP	840	4652	329	2078	413	2119	231	1510	189	1340
Designite	1739	4568	490	2179	510	2161	255	1511	222	1344
SonarQube	856		330		468		232		199	
Combined	832	3556	329	1893	413	1858	229	1316	189	1248

Table 8: Number of modules (#M) and units (#U) evaluated per project and tool

There are some mismatches between module and unit entries between our framework metric results and benchmark metric results. The differences between Designite and our framework are caused by implementation choices of both tools. Firstly, LAMP only evaluates top-level module members, which leads to the absence of anonymous class declaration units and inner module units compared to Designite. Secondly, LAMP evaluates enum class units, while Designite chose not to evaluate enum class units. Lastly, LAMP only categorises files as modules when it contains a declared module, while SonarQube includes all files as modules.

Besides several implementations choices, we had some difficulties matching the correct unit result of our LAMP framework and a benchmark unit metric result due to the possibility that there could be multiple units in a module with the same name. This duplication of names can occur when units are overloaded through parameter types or number of parameters. To solve the possible differences that occur due to mismatched units, we chose

[16] `https://github.com/airbnb/lottie-android`

to group units by name and sum the metric results. This allows us to detect differences on the unit level, but it might prevent us from spotting small differences in individual units with the name.

6.3.4 RxJava Results

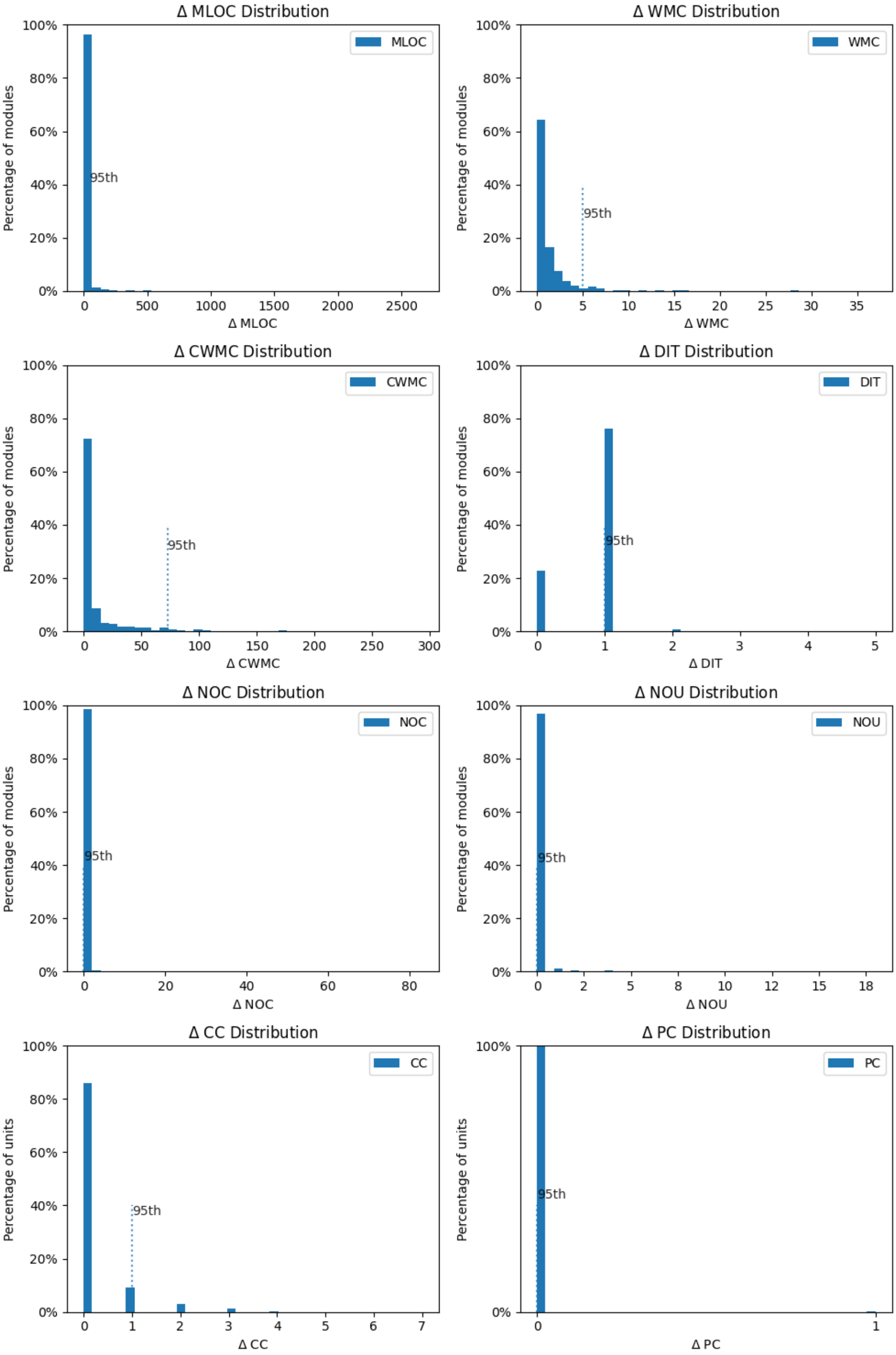


FIGURE 28: *RxJava* computation difference distribution per metric

The first project that was evaluated is RxJava. The distribution of differences is shown in Figure 28. The percentage of exact matches (i.e., delta (Δ) is zero) is listed in Table 9.

It is obvious that MLOC, NOC, NOU, and PC compute relatively similar metric results compared to the benchmark results. WMC, CWMC, and CC follow a negative exponential distribution. The CC score is generally higher than WMC because WMC is the sum of all units in a module. Therefore, WMC will produce higher differences. DIT shows low precision, while it generally is only a difference of one.

Metric	**$\Delta = 0$ (%)**
MLOC	9.38%
WMC	64.42%
CWMC	44.11%
DIT	22.72%
NOC	96.88%
NOU	97.00%
CC	85.91%
PC	99.83%

Table 9: Percentage of exact matches in *RxJava* metric results

Although exact matches give us a good indication of the extent to which our framework follows the same computation instructions, we must also look at the magnitude of difference between our prototype and the benchmark.

Metric	**Δ at 90th**	**Δ at 95th**	**Δ at 97.5th**	**Δ at 99th**	**Δ Max**
MLOC	26	45	95	341	2661
WMC	3	5	7	13	37
CWMC	42	72	109	172	293
DIT	1	1	1	1	5
NOC	0	0	1	3	83
NOU	0	0	1	2	18
CC	1	1	2	3	7
PC	0	0	0	0	1

Table 10: Metric result Δ values at several percentiles for *RxJava*

In Table 10, several percentiles are shown that indicate what the largest delta value is of that percentile. Analysing the metric differences per percentile, we see that the differences of WMC, DIT, NOC, NOU, CC, and PC are still small at higher percentiles. MLOC and CWMC show larger increments. The reasons for these values are summarised in Section 6.3.9.

6.3.5 Jenkins Results

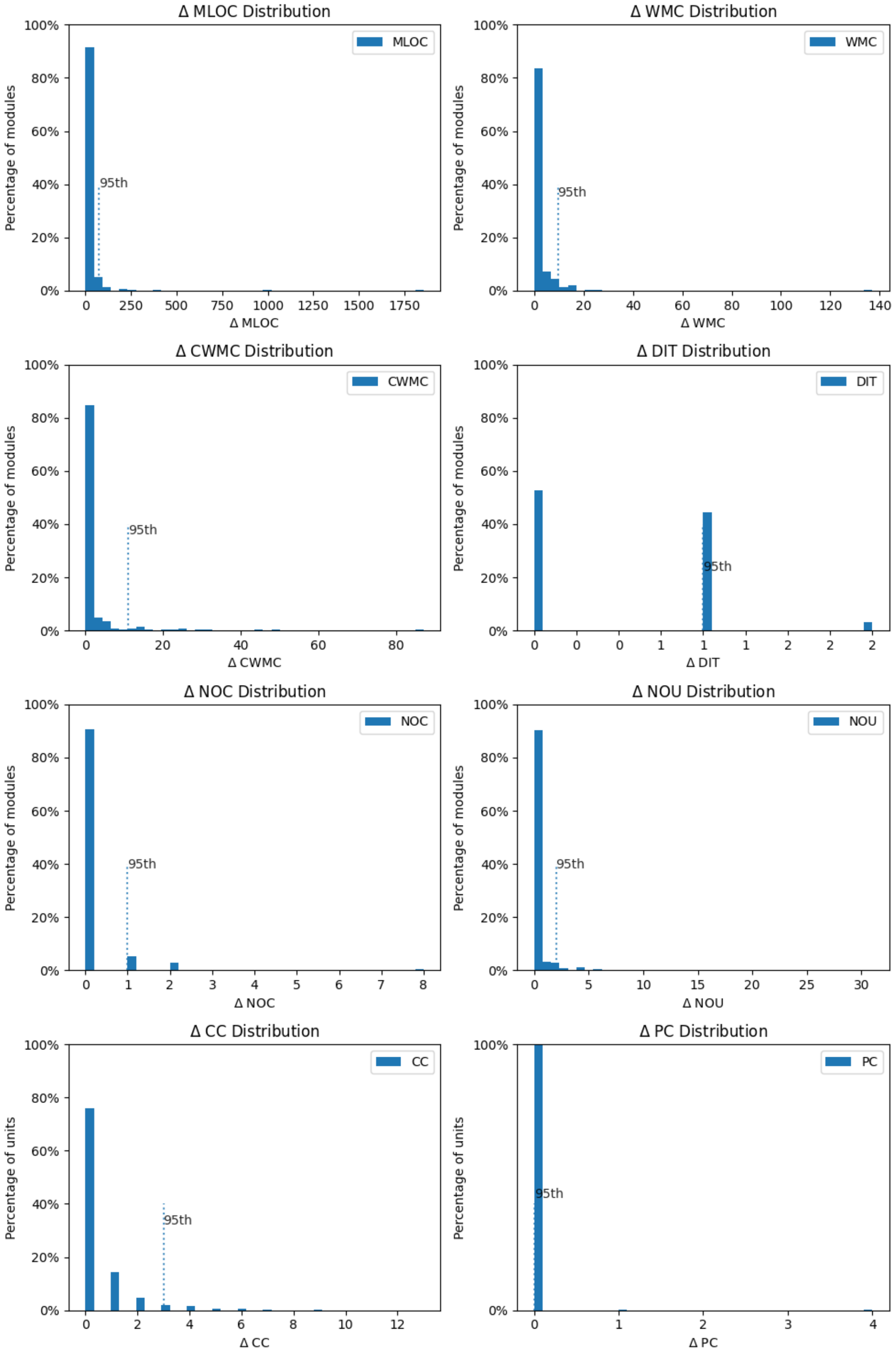


FIGURE 29: *Jenkins* computation difference distribution per metric

The second evaluated project is Jenkins. Jenkins largely shows similar results to RxJava, although it was able to predict the DIT metric a bit better. It follows the same distribution patterns as RxJava, indicating that the metrics are being computed consistently.

Metric	$\Delta = 0$ **(%)**
MLOC	3.65%
WMC	46.20%
CWMC	78.12%
DIT	52.58%
NOC	90.58%
NOU	90.27%
CC	75.81%
PC	99.74%

Table 11: Percentage of exact matches in *Jenkins* metric results

When we analyse the percentages of exact matches from Figure 11, we find that MLOC has a very low percentage. As described in RxJava, WMC performs worse than CC. The other metrics perform pretty well. CWMC computes more matches than it did for the RxJava project.

Metric	Δ **at 90**$^{\mathbf{th}}$	Δ **at 95**$^{\mathbf{th}}$	Δ **at 97.5**$^{\mathbf{th}}$	Δ **at 99**$^{\mathbf{th}}$	Δ **Max**
MLOC	40	74	111	211	1855
WMC	6	9	16	17	137
CWMC	5	11	21	30	87
DIT	1	1	2	2	2
NOC	0	1	2	2	8
NOU	0	2	3	4	31
CC	1	3	4	5	13
PC	0	0	0	0	4

Table 12: Metric result Δ values at several percentiles for *Jenkins*

As described in Table 12, the maximum values calculated in each percentile are relatively low for DIT, NOC, NOU, CC, and PC. For MLOC, WMC and CWMC, the differences are larger, although still far away from the maximum difference for the metric.

6.3.6 Mockito Results

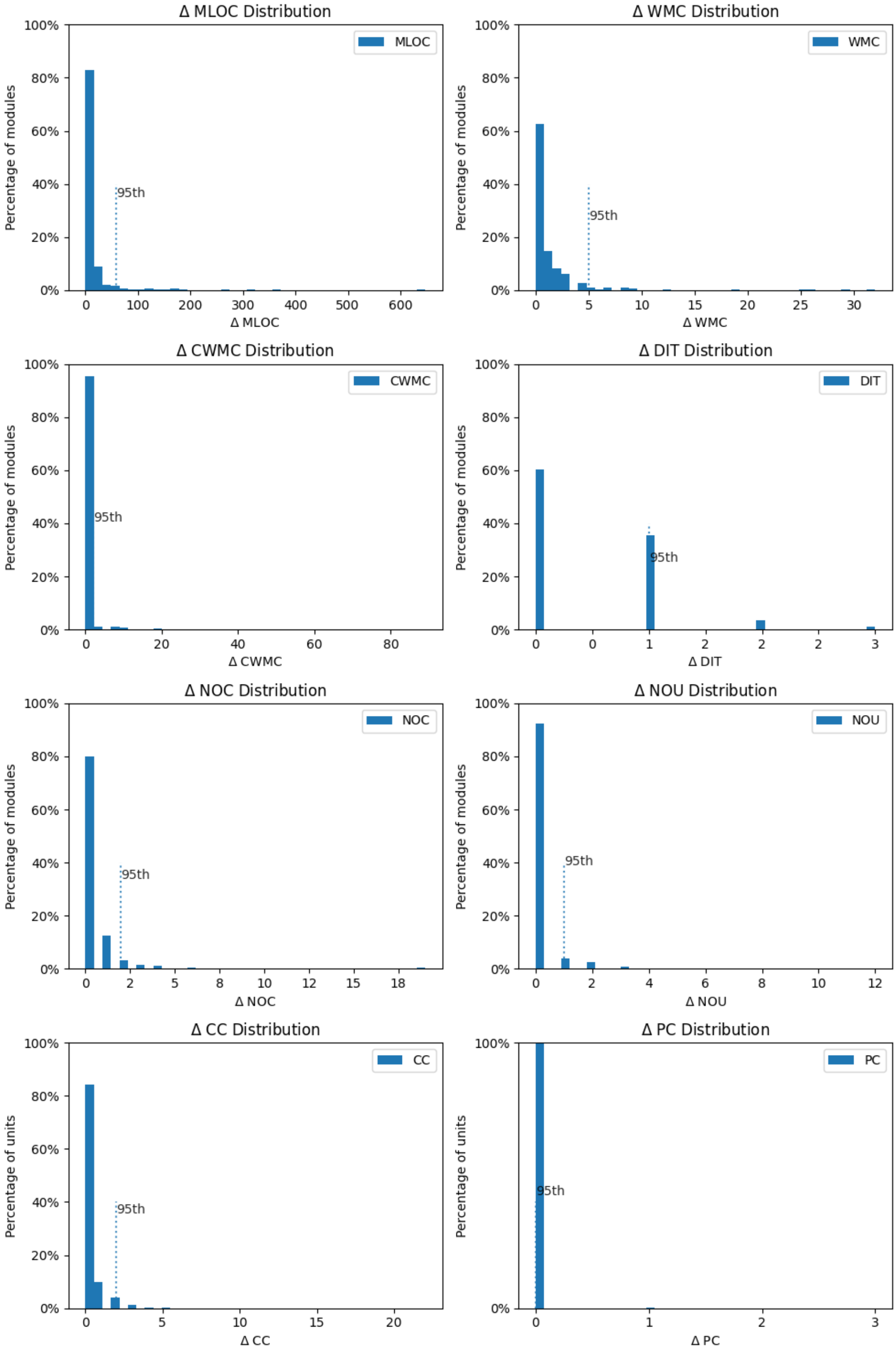


FIGURE 30: *Mockito* computation difference distribution per metric

When analysing the Mockito distributions in Figure 30, we find similar results to those seen with Jenkins. Again, the DIT metric shows improvement with a more linearly decreasing distribution.

Metric	$\Delta = 0$ (%)
MLOC	23.73%
WMC	62.47%
CWMC	91.53%
DIT	60.29%
NOC	79.90%
NOU	92.25%
CC	84.18%
PC	99.78%

Table 13: Percentage of exact matches in *Mockito* metric results

In general, every metric shown in Table 13 performs better compared to Jenkins. We see a higher percentage of exact matches for MLOC, well above the percentages seen in RxJava and Jenkins. CWMC also performs very well in the Mockito project compared to the previously analysed projects.

Metric	**Δ at 90$^{\text{th}}$**	**Δ at 95$^{\text{th}}$**	**Δ at 97.5$^{\text{th}}$**	**Δ at 99$^{\text{th}}$**	**Δ Max**
MLOC	26	58	126	176	647
WMC	3	5	8	12	32
CWMC	0	2	8	12	89
DIT	1	1	2	2	3
NOC	1	2	3	6	19
NOU	0	1	2	2	12
CC	1	2	2	3	22
PC	0	0	0	0	3

Table 14: Metric result Δ values at several percentiles for *Mockito*

Due to the higher percentages of exact matches, we see lower difference values at higher percentiles for almost all metrics in Table 14.

6.3.7 ANTLR4 Results

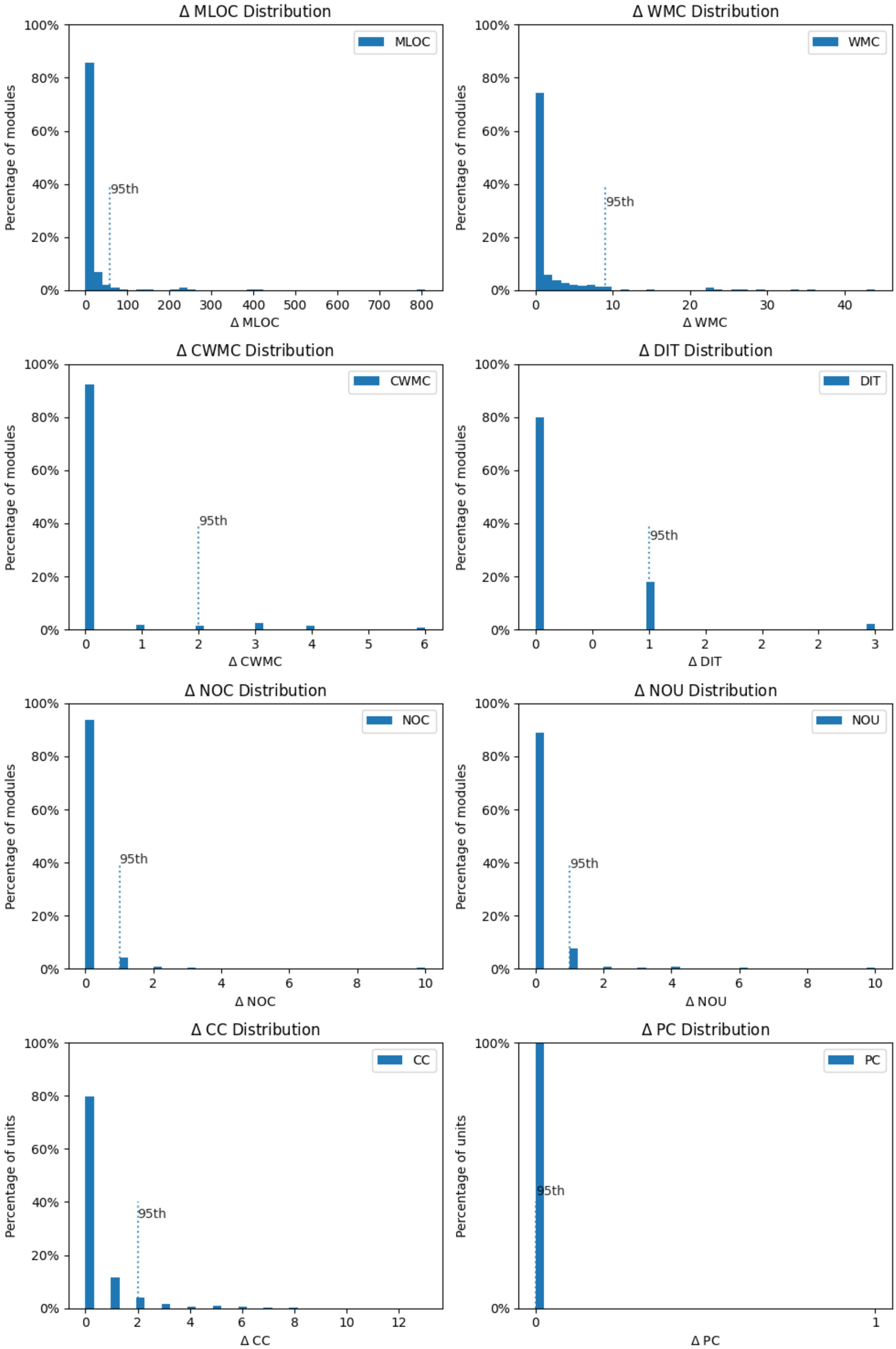


FIGURE 31: *ANTLR4* computation difference distribution per metric

As depicted in Figure 31, ANTLR4 shows a nice pattern similar to the Mockito project. It contains mostly nice distributions with a large percentage of values around zero. Again, the DIT metric shows better results compared to previous projects. This can occur due to low usage of inheritance.

Metric	$\Delta = 0$ **(%)**
MLOC	15.72%
WMC	63.32%
CWMC	92.14%
DIT	79.91%
NOC	93.89%
NOU	89.08%
CC	79.71%
PC	100.00%

Table 15: Percentage of exact matches in *ANTLR4* metric results

The number of exact matches in Table 15 confirms what we have seen previously in Figure 31, it performs quite well. Only MLOC cannot establish a large group of zero difference values.

Metric	**Δ at 90th**	**Δ at 95th**	**Δ at 97.5th**	**Δ at 99th**	**Δ Max**
MLOC	26	53	212	257	808
WMC	6	9	24	29	44
CWMC	0	2	3	4	6
DIT	1	1	1	3	3
NOC	0	1	1	2	10
NOU	1	1	2	4	10
CC	1	2	4	6	13
PC	0	0	0	0	0

Table 16: Metric result Δ values at several percentiles for *ANTLR4*

Due to the high number of exact matches within the ANTLR project, we see small values at higher percentiles in Table 16, following the distributions seen in Figure 31.

6.3.8 Lottie Results

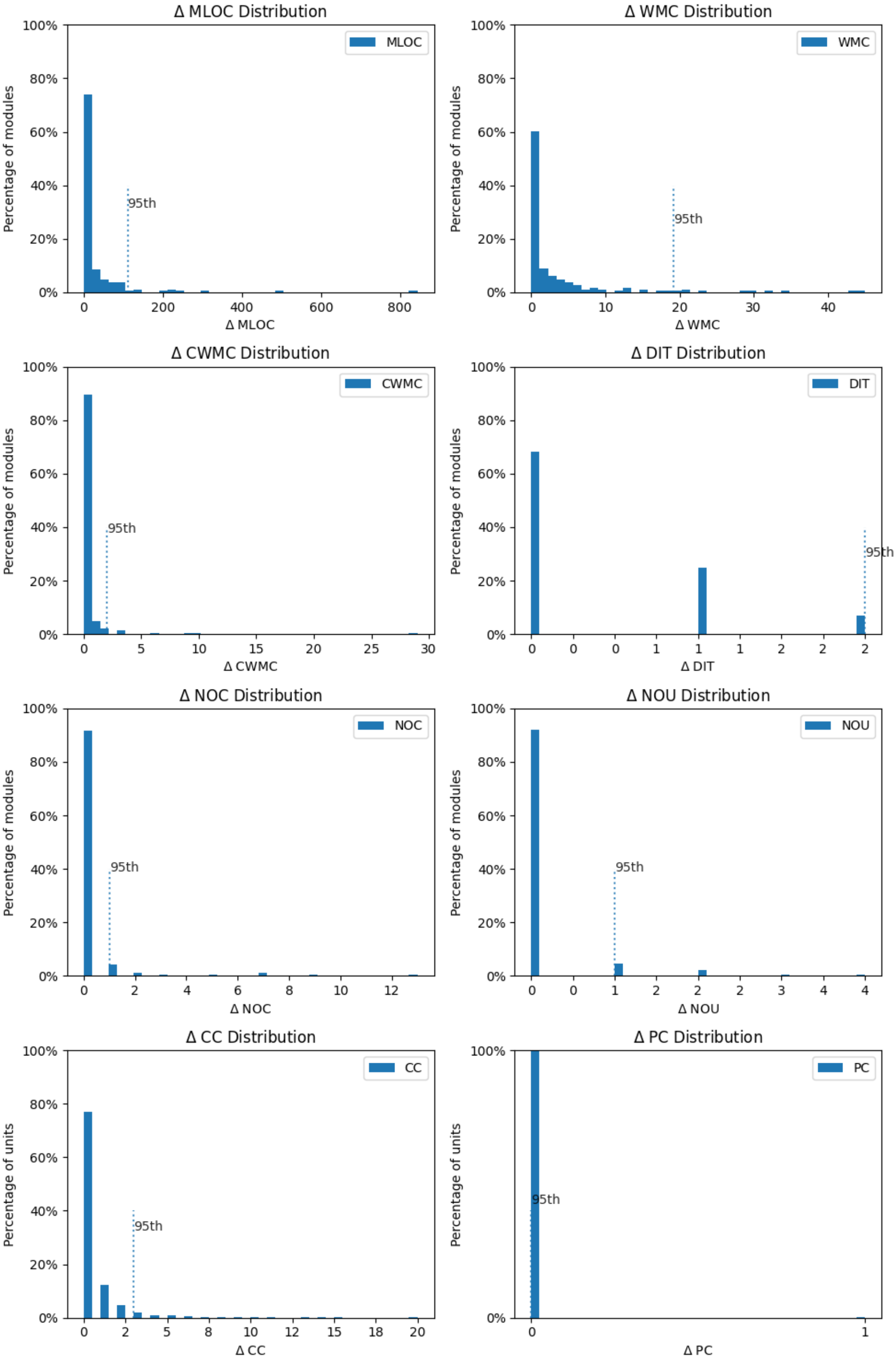


FIGURE 32: *Lottie* computation difference distribution per metric

The distribution of the metrics in Figure 32 follows the same distribution patterns as seen in the other projects. MLOC, WMC, CWMC and CC show a visible exponential distribution, while DIT looks linear. NOC, NOU, and PC show a large presence of values around zero.

Metric	$\Delta = 0$ (%)
MLOC	24.87%
WMC	44.97%
CWMC	89.42%
DIT	68.25%
NOC	91.53%
NOU	92.06%
CC	76.92%
PC	99.92%

Table 17: Percentage of exact matches in *Lottie* metric results

The number of exact matches in Table 17 gives a good indication of the precision and correctness between our prototype implementation and the benchmark. Again, CWMC can be calculated effectively, as well as NOC, NOU, and PC. Although MLOC is low from a percentage point of view, it is still very high compared to the other analysed projects.

Metric	**Δ at 90$^{\text{th}}$**	**Δ at 95$^{\text{th}}$**	**Δ at 97.5$^{\text{th}}$**	**Δ at 99$^{\text{th}}$**	**Δ Max**
MLOC	73	105	221	300	844
WMC	10	18	29	34	45
CWMC	1	2	3	9	29
DIT	1	2	2	2	2
NOC	0	1	3	7	13
NOU	0	1	2	2	4
CC	2	3	5	8	20
PC	0	0	0	0	1

Table 18: Metric result Δ values at several percentiles for *Lottie*

The maximum differences at different percentiles in Table 18 give a good indication that the prototype and benchmark results do not deviate significantly for most metrics. MLOC and WMC show larger differences, although in the case of WMC this is due to the sum of CC.

6.3.9 Metric Result Analysis

We have seen the results of the difference distribution for each analysed project, their percentage of exact matches, and the maximum differences in several percentiles. As described in Section 6.3.1, we wanted to analyse all metric outliers for each project. During this analysis, several reasons were found throughout the projects that caused the differences seen in the results. In this section, we give, for each metric, one or more reasons for the differences between our LAMP prototype and the benchmark tooling.

Although the figures for the analysed projects show that there are still some differences between our prototype and the benchmark tools, we found that these differences are often not due to incorrectness of our framework, but rather implementation choices. These implementation choices could be implemented into the framework because the necessary information is contained within the metamodel. In some cases, our framework prototype outperforms benchmark tools due to bugs in the benchmark tools.

MLOC

- Designite contains a bug where, for an arbitrary number of modules, MLOC computes a value of zero. This was detected on some large outliers that had a lot of documentation in the module.
- Designite counts comment lines for the MLOC metric, where the LAMP prototype does not count comment lines as code lines. This explains the large differences and low number of exact matches in our results. Having referred back to our definition of LOC given in Section 2.4.1, we conclude that our prototype currently excludes comment lines and blank lines in the computation of MLOC. Using the metamodel design, it would be possible to count MLOC using a module's start and end line. This would include blank lines, which are excluded in the computation by Designite.
- The LAMP prototype misses LOC counts when LogicalSequence operators are placed on a separate line by itself since the metamodel only stores operand elements, not operator elements. This limitation in computation can be resolved by including the operator as an element in the logical sequence.

WMC

- The LAMP prototype does not include inner, nested, and anonymous classes in the WMC count. For a number of modules in projects that heavily relied on those types of classes, the count of WMC was therefore much lower.

CWMC

- The LAMP prototype does not include nested, inner, and anonymous class declaration CWMC computations in the CWMC count of the outer module. This computational instruction was not part of the SonarSource specification [51], but was implemented for the implementation of this specification in SonarQube.

DIT

- The LAMP prototype does not include interfaces in the calculation of DIT, while Designite does include it. Therefore, there was a large portion of outliers that contained a difference of one, due to the inclusion of interfaces in the benchmark computation.
- Designite does not include extended modules that are imported from external dependencies. While the LAMP prototype is not able to traverse the external dependency inheritance trees, we do increment DIT for such externally imported modules by one. We argue that every dependency, internal or external, attributes to the depth of an inheritance tree.

- Designite does not count DIT for class extensions that reference a class in the same file. In he LAMP prototype, every top-level declared module inside a file becomes its own entity and therefore it does count these modules as an increment.

NOC

- Designite does not count the NOC for abstract module declarations. The LAMP prototype does include abstract modules in its computation of NOC.
- The LAMP prototype does not count the NOC for interface modules while Designite does count these.

NOU

- Designite excludes enum classes from their NOU results. The LAMP prototype sees enums as modules because they can include complex code, although it is generally not the case.
- The LAMP prototype does not include units of inner, nested and anonymous modules for the NOU count of the outer module.

CC

- Designite does not count CC for the occurrence of a ternary, else or a logical operator, which is done by LAMP. We argue that our framework implementation for CC is the correct way to evaluate CC because our selected languages use short-circuit evaluation. Due to this difference, the percentage of exact matches was much lower in each project.

PC

- Designite contains a bug where one too many parameter is counted for a small number of arbitrary units.

6.3.10 Metric Precision Analysis

In Table 19, the percentages of exact matches for each project are shown side by side. As we have explained in the previous section, there are many differences in some of the metrics that we included in our evaluation. This impacts the percentage of exact matches, but we were able to show these differences, which provides information about the capability of the LAMP metamodel to capture important metric requirements.

Metric	RxJava	Jenkins	Mockito	ANTLR4	Lottie	Weighted Avg.
MLOC	9.38%	3.65%	23.73%	15.72%	24.87%	13.61%
WMC	64.42%	46.20%	62.47%	63.32%	44.97%	59.03%
CWMC	44.11%	78.12%	91.53%	92.14%	89.42%	69.38%
DIT	22.72%	52.58%	60.29%	79.91%	68.25%	46.34%
NOC	96.88%	90.58%	79.90%	93.89%	91.53%	91.47%
NOU	97.00%	90.27%	92.25%	89.08%	92.06%	93.52%
CC	85.91%	75.81%	84.18%	79.71%	76.92%	81.68%
PC	99.83%	99.74%	99.78%	100.00%	99.92%	99.84%

Table 19: Summary of exact matches across all evaluated projects

From the comparison of the values seen in Table 19 and the reasons behind the differences, we can extract some interesting points. MLOC is largely influenced by the number of comments that are present in the modules. We can argue that we can, to some extent, guess the amount of documentation present in each project. A smaller number of exact matches would indicate a larger amount of documentation. By looking at the large difference between LAMP and Designite in analysed modules in Table 8, we can derive that RxJava and Jenkins contain a large number of inner, nested and anonymous classes. This impacts the WMC and CWMC metrics. RxJava contains many interfaces, which impacts the DIT metric more drastically than the other projects. The lower exact matches in CC for Jenkins could indicate that Jenkins uses more constructs such as logical operators and ternary expressions. We have not analysed whether these assumptions are valid.

6.4 Evaluation Findings

For every selected metric, we were able to provide a mapping of the requirements to calculate the metric with one or more language-agnostic source elements. Some metrics require us to traverse multiple ModuleRoots to capture inheritance information or references to source elements outside of one ModuleRoot. This information can be extracted with some workarounds, although there were some limitations with regard to type resolution. We must be cautious of the procedures that are defined for type resolution on a language-agnostic level. While type resolution in one programming language is complex but clearly defined, we must know the behaviour of constructs of all languages individually on a language-agnostic level. This requires us to model behaviour of each language explicitly to contain behavioural information in a language-agnostic representation. Therefore, transformation conventions from individual languages to a language-agnostic representation must be described to produce uniform results in the evaluation phase of the framework.

For each project, we compared the results from our framework prototype with the benchmark results. This comparison shows that our framework can correctly compute the code quality metrics on a language-agnostic level for the metrics LOC, CC, WMC, CWMC, DIT, NOC, PC, and NOU. Our framework did not include constructs that were not top-level members of modules or inner modules, resulting in a difference in the computed results between our implementation and the benchmarks. Aside from these implementation choice differences, we can conclude that the metrics computed by the framework prototype deviate only by small amounts from language-specific tools. We found that we can capture all the required constructs to calculate the metrics while measuring it on a language-agnostic level. As a result, the framework can be considered more versatile than language-specific tools because of its support for multiple languages.

7 Related Work

This thesis was part of a research line at Info Support on multi-paradigm code quality. Landkroon researched fault-prediction of the MP programming language Scala [11]. It outlines the MP constructs present in Scala. He used three popular Scala projects on GitHub to test if the faults present in their issue tracker could be a predictor for bugs by using code quality metrics. Landkroon also developed a new validation methodology that improves upon Briand's Validation Methodology [68]. Zuilhof developed MP metrics for C# [12]. This research outlines the MP constructs present in C#. Konings adopts these developed MP metrics within Scala [13]. These theses concluded that accurate predictions were lacking due to data constraints, which inspired this thesis to solve that constraint. Furthermore, several metrics were used within these theses, which proved useful in determining the maintainability of the source code.

Regarding maintainability, the Software Improvement Group has designed the language-agnostic SIG Maintainability Model (MM) [3]. This model measures all sub-characteristics of maintainability by mapping them to language-agnostic source code metrics. Using this mapping, they determine a maintainability score of a source code. In this research, we adopted the terminology of SIG and many mappings from metrics to quality characteristics. We were unable to use their tool called Sigrid [69] that implemented the MM because it was not commercially available.

There are tools available for metaprogramming. srcML is an XML format for the representation of C/C++/Java source code that wraps source code with information from the AST into a single XML file. It can be used to perform scalable lightweight fact-extraction and transformation since other XML tools can be used once it is in the XML form (e.g., XPath and XSLT) [70]. The tool still holds some language-specific structural information, which denies us the possibility to extract metric results in a uniform way.

Another tool for source code analysis is the meta-programming language Rascal [43]. It can be used to extract metrics from programming languages and is extendable with libraries such as M^3 [64, 71]. The M^3 model is an extensible language-agnostic model for capturing facts about source code for future analysis [64]. Due to a lack of documentation online, we chose not to use Rascal or its M3 extension.

The Object Management Group (OMG) has designed the Abstract Syntax Tree Metamodel (ASTM) standard. ASTM itself serves as a universal high-fidelity gateway for modelling code at its most fundamental syntactic level [72]. This document also outlines the core concepts of imperative programming languages in the Generic Abstract Syntax Tree Metamodel (GASTM). The GASTM allowed us to look differently at programming languages and their associated grammars. The reference of source elements and scopes helped us better understand how these languages were structured. We did not copy much more from this metamodel because it was too generalised and contained a lot of information that was not required for our purposes. With regards to multi-paradigm modelling, Owens [73] describes an extension of GASTM for the functional paradigm. Functional operations on Lists, Lambda functions and support for containerless list of functions and constants are included in the extension. These extensions were given as a research direction in his research, which allowed us to take those concepts and try to relate them to our metamodel.

8 Conclusion

This section concludes the research of a language-agnostic multi-paradigm code quality assurance framework. We began this thesis by stating the importance of abstracting away from language-specific tooling in code quality assurance to relieve ourselves from the data constraints that such an approach imposes. We have analysed four major MP programming languages, namely Java, C#, Scala, and Kotlin. With that analysis, we were able to pinpoint the important constructs that could be used to create a language-agnostic representation of MP programming languages. We designed a metamodel which was integrated into the workflows of our code quality assurance framework. To validate the correctness of this metamodel and the framework, a prototype was developed that was in accordance with these designs. This prototype was evaluated by comparing the metric computations with two benchmarks. This leads to the conclusion of this thesis in this section.

In Section 8.1, the findings of all the research questions are presented. In Section 8.2, the main research question is answered. We discuss the impact and design decisions of our framework in Section 8.3. Finally, we provide future research directions in Section 8.4.

8.1 Findings

Throughout this document, we have explored the possibilities of measuring properties of multi-paradigm languages in a language-agnostic manner. In this section, we will summarise our findings by answering our main research question and subquestions.

RQ1 *What are the constructs of multi-paradigm programming languages?*

Before constructs for multi-paradigm languages could be accurately identified, the analysis of paradigm concepts was done for the object-oriented paradigm and the functional paradigm. For a language-agnostic analysis, we selected four programming languages that can be seen as multi-paradigm languages within our definition. Using paradigm concepts, we analysed these four languages to identify implementation differences and commonalities. In this analysis, it was concluded that all languages embraced the concepts of the object-orientated paradigm in its entirety, while the functional concepts were not supported to that extent. Due to mutable state within the object-oriented paradigm and immutability as cornerstone of the functional paradigm, it is evident that multi-paradigm languages are not functional, but merely functionally styled. The inclusion of functional constructs such as lambda functions, functions as types, and pattern matching makes object-oriented source code more declarative, but the multi-paradigm languages are still imperative.

RQ2 *Which code quality characteristics must the framework capture, and how?*

One of the primary goals of having an idea of the quality of code is the identification of potentially problematic parts within your software. Therefore, our goal was to focus on the developer who must be able to develop and maintain multi-paradigm source code. We adopted the definition of maintainability by ISO/IEC 25010:2011 and used the GQM approach to connect our goal to specific quality characteristics and related metrics. Well-known and validated metrics from the literature were used to identify the degree of code quality for each subcharacteristic. These metrics provide quantitative information on the source code, unit, module, and coupling levels. A missing piece of information to answer the GQM questions is threshold values for most metrics. Not every metric has publicly available threshold values that can tell us if the metric results can be deemed bad for the maintainability subcharacteristic. It is also unknown to what extent each metric has an impact on a particular subcharacteristic, as the literature often states that, in general, high values impact complexity or maintainability.

RQ3 *How can we measure code quality in a language-agnostic multi-paradigm manner?*

To measure code quality at a language-agnostic level, we had to use the analysed constructs of RQ1 and see how they could be generalised so that we could compute the metrics that we defined in RQ2. For that reason, we designed the LAMP metamodel, which is an abstraction of multi-paradigm syntax trees. The LAMP metamodel summarised key constructs of the selected MP programming languages using language-agnostic terminology. The metamodel has three important requirements. It must model all semantic elements within the source code, but only metric-sensitive information must be explicitly typed. It must preserve the contextual information of the source code elements, including its source location. It must have declarable return type information for type resolution. With these requirements, the metamodel was designed and theoretically capable of measuring all defined metrics.

RQ4 *How does this framework perform in comparison to language-specific analyses?*

All metrics were evaluated to determine the ability of the metamodel to map source elements onto the metric requirements. The coupling metrics required advanced type resolution that is currently not implemented in the framework, although it is achievable with the information contained in the metamodel. In our evaluation, we were able to provide language-specific benchmarks for the metrics LOC, CC, WMC, CWMC, DIT, NOC, PC, and NOU. As shown in our evaluation, there are several differences between the prototype metric results and the benchmark metric results. The majority of differences were caused by the implementation choices being different between the prototype and the benchmark. Aside from these implementation choice differences, we can conclude that the metrics computed by the framework prototype deviate only by small amounts from language-specific tools. We found that we can capture all the required constructs to calculate the metrics while measuring it on a language-agnostic level. As a result, the framework can be considered more versatile than language-specific tools because of its support for multiple languages.

8.2 Final Thought

This thesis describes the framework for computing code quality metrics for MP programming languages in a language-agnostic manner. We proposed a design of a language-agnostic metamodel that captures the most important paradigm constructs of MP programming languages, and we implemented a prototype of our framework to validate that measuring code quality on a language-agnostic level is possible. We show that our metamodel is theoretically able to capture all constructs required for computing well-known code quality metrics. While the prototype still lacks important type resolution functionality, the proposed coupling metrics can be measured with our framework. With our benchmark evaluation, we also show, in practise, that our framework is able to compute a subset of the metrics accurately.

We conclude that our LAMP metamodel is capable of capturing the most important constructs of MP programming languages, and our framework is able to put forward a consistent workflow to assure code quality. Therefore, we argue that our framework design represents a significant step towards solving the data scarcity problem with fault proneness detection in MP programming languages.

8.3 Discussion

During our research, many aspects had to be taken into account, which illustrates the complexities of creating an abstraction from several programming languages. In the following paragraphs, we will discuss our design choices and highlight several threats to the validity of this research.

Choice of languages: The choice was made to select four major programming languages that adopted functionality that labelled them as multi-paradigm. In our cross-comparison, we analysed these languages, and it showed many similarities and also some differences. It is yet unclear if the selected languages give a good representation of all multi-paradigm programming languages. It can be argued that these languages have been arbitrarily chosen, although all rank highly on the Popularity Index [2]. For the selected languages, the choice was made to stick to their latest stable version, which included many functionally styled constructs compared to the older version. During our evaluation, we found that most mature source code projects in Java do not use newer functionally styled constructs often.

Metamodel portability: During our analysis of each selected programming language, we chose to stick closely to paradigm concepts to ensure that we were not constraining our language-agnostic representation to an individual programming language. When including new programming languages, we must stick to these paradigm concepts to port this new programming language to the language-agnostic representation. In Section 4.8, we designed Expression which allows us to have a lot of freedom in expressing constructs from any language. Therefore, when porting to new languages, we can model unknown constructs. This decision has yet to be validated in practice to know if it is possible with the current metamodel design.

Recognising MP smells: During our research on code metrics and smells, there were some interesting OOP code smells that cannot be seen as a code smell in a multi-paradigm setting. Data class is a code smell that says that a class with exclusively fields and corresponding getters and setters is bad, while multi-paradigm languages have implemented a record, data class, or case class that are exactly that. The "switch statements" smell suggests that the developer should avoid using switch statements and use polymorphism, but the upcoming of a range of pattern matching possibilities contradicts this smell. It is important to think again about some patterns that were initially considered a smell, as strengths of combining multiple paradigms.

Modelling choices: Our metamodel design stays close to the semantic structure of our selected programming languages. It provides the freedom to transform any construct into a metamodel element, explicitly with a defined element, or implicitly with providing a language-specific context. As a result of this design, all selected metrics could be computed in theory. During the creation of our prototype, this theory was put into practice. We were unable to show for all metrics what was required to do computations correctly. Coupling metrics such as LCOM, RFC, CBO and LSE require complex operations on the symbol tree to resolve advanced types from other parts of the project.

Transformation conventions: Implementing transformations from language-specific representations to the metamodel representation requires us to conform to a certain structural composition, which is not covered explicitly by the metamodel. In the metamodel, constructs can be freely modelled using Expression elements and its inner scope SourceElement items. Due to this freedom in transformation, there are multiple ways to model a language, and thus multiple ways in which a metric could be computed. Therefore, it is crucial to have transformation conventions to streamline the transformation process for each MP programming language.

Transforming language behaviour: When abstracting, we must take into account the way that a language is compiled. Although language constructs can look identical on a syntactical level, they can have different behaviours. Therefore, the framework should take the behaviour of the language into account during the transformation process. This adds some complexity to the framework, which language-specific approaches do not face. We assume that transforming behaviour is not difficult as long as it is done consistently. Once the transformation for a language has been created, it can be used by anyone.

Type resolution: To measure coupling in source code at the metamodel level, the types must be fully integrated into the metamodel. In our model, this was partly out of scope due to the exclusion of type parameterisation. However, all type information must be available to implement type resolution correctly.

Shifting the implementation burden: This design of a language-agnostic representation for multi-paradigm languages results in a shift of load on developers from developing language-specific tools to developing language transformations once. It can be argued that the current general-purpose design of the metamodel adds little value and much unnecessary complexity to the quality assessment, where it would be more convenient to implement a language-specific tool. We think this would be an incorrect argument since we showed, using the metrics that were computed, that we can measure a lot of code quality metrics with our language-agnostic approach. We think that the implementation burden for creating transformations for every language, considering described transformation conventions, does not outweigh the benefit of having a larger data set on which to base future analysis.

8.3.1 Threats to Validity

There are two threats to validity identified for this thesis. We cover them in the following paragraphs.

Correctness of prototype: Due to time constraints, we were unable to test all phases of our prototype. By evaluating the computed metrics and comparing them with benchmarks, we are able to evaluate the correctness of the prototype. When the computed metrics match the benchmarks in a large test set, the implementation of the underlying metric computations is assumed to be correct as well. We cannot assume that the transformations from a language-specific representation are correct in its entirety because we only compared a subset of our metrics with the benchmark data.

Modelling bias: We analysed multiple programming languages and extracted their constructs which were modelled in a metamodel. Due to our iterative approach to creating the metamodel, we adapted the metamodel during the development of transformations of Java constructs. Therefore, we must acknowledge a possible bias towards a structure that looks similar to the Java grammar. We demonstrate our effort to eradicate this bias as much as possible by also keeping other language constructs in mind.

8.4 Future Work

In this thesis, we have identified a range of interesting research directions for future work. This section covers the importance and opinions on these future work items.

Additional paradigms: Although this thesis focuses on the combination of OOP and FP, other paradigms can be interesting to integrate into a generic representation. When focusing more on data flow within source code, it can be interesting to include the reactive paradigm, which captures data streams and propagation of change. This can be interesting to analyse and combine it with for comprehensions. Furthermore, data flow analysis can be enriched with the concurrent paradigm, to model more complex behaviour of multi-threaded data flows. We assume that the addition of these paradigms would push the metamodel into the direction of modelling call graph information.

Pattern matching: Currently, there are not many metrics related to pattern matching in addition to size and variable use [10]. A visible trend for adoption within the selected languages is the addition of new types of pattern matching functionality. Future research could be focused towards measuring code quality of these patterns within a language, or even on a language-agnostic level.

Code smell detection: The current prototype is focused on computing metrics that directly relate to code quality. Some of these code metrics are required for detecting anti-patterns within a code base. We have covered several candidate code smells that could be detected with such metrics, namely Mutating Lambda Function, Data Class, Message Chains, and Lack of Sequence Comprehension. In the future, it can be extended with detecting these anti-patterns within the metamodel implementations.

Language-agnostic fault proneness detection: Assuming that the computed metrics of the prototype are correct and that there are transformations available for multiple MP programming languages. We can research and develop a language-agnostic fault proneness detection tool, as previously done in a language-specific manner by Zuilhof and Konings [12, 13]. Due to the accessibility of more training data, these prediction models can be better trained.

Quality improvement suggestions: This research has not yet focused on the assurance aspect of our framework. More research on metric thresholds, quality improvement suggestions, and the presentation of those suggestions is required to allow code quality assurance within our framework.

References

[1] Peter Van Roy et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104:616–621, 2009.

[2] Pierre Carbonnelle. PYPL popularity of Programming Language index. `https://pypl.github.io/PYPL.html`. Accessed: 04-07-2022.

[3] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A Practical Model for Measuring Maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, 2007.

[4] Søren Pedersen. The impact of Poor Software Quality, Aug 2021.

[5] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time.* O'Reilly Media, 2020.

[6] ISO/IEC 25010. ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Technical report, ISO/IEC, 2011.

[7] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[8] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Inc., USA, 1977.

[9] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[10] Chris Ryder and Simon Thompson. Software Metrics: Measuring Haskell. In Marko van Eekelen, editor, *Trends in Functional Programming*, Trends in Functional Programming. Intellect Books, Bristol, UK, September 2005.

[11] Erik Landkroon. Code quality evaluation for the multi-paradigm programming language Scala. Master's thesis, Universiteit van Amsterdam, 2017.

[12] Bart Zuilhof, Rinse van Hees, and Clemens Grelck. Code Quality Metrics for the Functional Side of the Object-Oriented Language C#. In *SATToSE*, 2019.

[13] S. Konings. Source code metrics for combined functional and Object-Oriented Programming in Scala. Master's thesis, University of Twente, November 2020.

[14] Marnick Q.T.P. van der Arend. Language-agnostic multi-paradigm code quality assurance framework. In *Proceedings of the Belgium-Netherlands Software Evolution Workshop, Mons, Belgium, September 12-13, 2022*, volume 3245 of *CEUR Workshop Proceedings.* CEUR-WS.org, 2022.

[15] Stefan L. Ram. Dr. Alan Kay on the meaning of "object-oriented programming". `https://www.purl.org/stefan_ram/pub/doc_kay_oop_en`, Jul 2003. Accessed: 04-07-2022.

[16] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications, Third Edition.* Addison-Wesley Professional, third edition, 2007.

[17] Martín Abadi and Luca Cardelli. A theory of objects. In *Monographs in Computer Science*, 1996.

[18] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[19] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. *SIGPLAN Not.*, 43(10):423–438, oct 2008.

[20] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[21] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[22] D. A. Spuler, A. S. M Sajeev, David A. Spuler, and A. S. M. Sajeev. Abstract Compiler Detection of Function Call Side Effects, 1994.

[23] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, sep 1989.

[24] William D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 174–185, New York, NY, USA, 1998. Association for Computing Machinery.

[25] Henk (Hendrik) Barendregt and E. Barendsen. Introduction to lambda calculus. *Nieuw archief voor wisenkunde*, 4:337–372, 01 1984.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[27] Microsoft. C# documentation. `https://docs.microsoft.com/en-us/dotnet/csharp/`, 2022. Accessed: 2022-09-01.

[28] Microsoft. C# Reference. `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/`, 2022. Accessed: 2022-09-01.

[29] IEEE. IEEE Standard for Software Quality Assurance Processes. *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, pages 1–138, 2014.

[30] David S. Alberts. The Economics of Software Quality Assurance. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, page 433–442, New York, NY, USA, 1976. Association for Computing Machinery.

[31] ISO/IEC. ISO/IEC 9126. Software engineering – Product quality. Technical report, ISO/IEC, 2001.

[32] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, USA, 2004.

[33] Norman E. Fenton and Martin Neil. Software Metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 357–370, New York, NY, USA, 2000. Association for Computing Machinery.

[34] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[35] Software Improvement Group (SIG) and TÜV Informationstechnik GmbH (TÜViT). SIG/TÜViT evaluation criteria - Trusted Product Maintainability, version 14.0, 04 2022.

[36] Joost Visser, Sylvan Rigal, Gijs Wijnholds, Pascal Van Eck, and Rob van der Leek. *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc.", 2016.

[37] Tiago Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. *IEEE International Conference on Software Maintenance, ICSM*, pages 1 – 10, 10 2010.

[38] Dennis Bijlsma, Miguel Ferreira, Bart Luijten, and Joost Visser. Faster issue resolution with higher technical quality of software. *Software Quality Journal - SQJ*, 20:1–21, 06 2012.

[39] Kaushal Bhatt, Vinit Tarey, Pushpraj Patel, Kaushal Bhatt Mits, and Datana Ujjain. Analysis of source lines of code (SLOC) metric. *International Journal of Emerging Technology and Advanced Engineering*, 2(5):150–154, 2012.

[40] Frances E. Allen. Control Flow Analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[41] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[42] Tim Sheard. Accomplishments and Research Challenges in Meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, pages 2–44, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[43] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, 2009.

[44] Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013.

[45] Darryl Owens and Mark Anderson. A generic framework for automated quality assurance of software models - application of an abstract syntax tree. In *2013 Science and Information Conference*, pages 207–211, 2013.

[46] Microsoft. The .NET Compiler Platform SDK. `https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/`, 2022. Accessed: 2022-09-01.

[47] Scalameta. Scalameta - Library to read, analyze, transform and generate Scala programs. `https://scalameta.org/`, 2022. Accessed: 2022-09-19.

[48] JetBrains Research. Kotlin Grammar. `https://kotlinlang.org/docs/reference/grammar.html`, 2023. Accessed: 2023-02-04.

[49] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

[50] Vu Nguyen, Sophia Deeds-rubin, Thomas Tan, and Barry Boehm. A SLOC Counting Standard. In *COCOMO II Forum 2007*, 2007.

[51] G Ann Campbell and SA SonarSource. Cognitive complexity. *SonarSource: Geneva, Switzerland*, 2020.

[52] Österberg Melker. Measuring Functional Purity In C#. Master's thesis, Uppsala University, September 2021.

[53] M. Fowler. *Refactoring: Improving the Design of Existing Code.* A Martin Fowler signature book. Addison-Wesley, 2019.

[54] Ammar Hamid and Vadim Zaytsev. Detecting refactorable clones by slicing program dependence graphs. In Davide Di Ruscio and Vadim Zaytsev, editors, *Post-proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2014, L'Aquila, Italy, 9-11 July 2014*, volume 1354 of *CEUR Workshop Proceedings*, pages 37–48. CEUR-WS.org, 2014.

[55] Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, and Heitor C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, 2012. Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.

[56] Tushar Sharma. Designite - A Software Design Quality Assessment Tool, May 2016.

[57] Paul E. Black. DADS: The On-Line Dictionary of Algorithms and Data Structures, 2020-09-17 2020.

[58] Gianluigi Caldiera, Victor Basili, and H Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, pages 528–532, 1994.

[59] Manny M Lehman and Laszlo A Belady. *Program evolution: processes of software change.* Academic Press Professional, Inc., 1985.

[60] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, 2013.

[61] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17:243–275, 06 2012.

[62] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, 2012.

[63] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36:20–36, 2010.

[64] Bas Basten, Mark Hills, Paul Klint, Davy Landman, Ashim Shahi, Michael J. Steindorfer, and Jurgen J. Vinju. M^3: A general model for code analytics in rascal. In Olga Baysal and Latifa Guerrouj, editors, *1st IEEE International Workshop on Software Analytics, SWAN 2015, Montreal, QC, Canada, March 2, 2015*, pages 25–28. IEEE Computer Society, 2015.

[65] S. Bradner. IETF RFC 2119: Key words for use in RFCs to Indicate Requirement Levels, 1997.

[66] Shudi (Sandy) Gao, C. M. Sperberg-McQueen, and Henry Thompson. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures, 2012.

[67] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[68] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[69] Software Improvement Group. Sigrid® | Software Assurance Platform. https://www.softwareimprovementgroup.com/solutions/sigrid-software-assurance-platform/, 2023. Accessed: 2023-03-07.

[70] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *Proceedings of the 11th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 173–184. IEEE Computer Society, 2011.

[71] Nico de Groot. Analysing and Manipulating CSS using the M^3 Model. Master's thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, July 2016.

[72] Object Management Group (OMG). Abstract Syntax Tree Metamodel (ASTM) Version 1.0. Standard, Object Management Group (OMG), January 2011.

[73] Darryl Owens. *A generic framework facilitating automated quality assurance across programming languages of disparate paradigms.* PhD thesis, Edge Hill University, Ormskirk, UK, 2016.

[74] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The Java® Language Specification. https://docs.oracle.com/javase/specs/jls/se17/html/, 2021. Accessed: 2022-09-23.

[75] Brian Goetz. Data Classes for Java - History. https://openjdk.org/projects/amber/design-notes/data-classes-historical-2, 2018. Accessed: 2022-09-29.

[76] Brian Goetz. Data Classes and Sealed Types for Java. https://openjdk.org/projects/amber/design-notes/records-and-sealed-classes, 2019. Accessed: 2022-09-22.

[77] Microsoft. Patterns - Pattern matching using the is and switch expressions, and operators and, or and not in patterns. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns#declaration-and-type-patterns, 2023. Accessed: 2023-02-04.

[78] Scala Center. Tour of Scala. https://docs.scala-lang.org/tour/tour-of-scala.html, 2022. Accessed: 2022-09-14.

[79] Scala Center. Scala 3 - Book. https://docs.scala-lang.org/scala3/book/introduction.html, 2022. Accessed: 2022-09-14.

[80] Simon Peyton Jones. How to make a fast curry: push/enter vs eval/apply. In *International Conference on Functional Programming*, pages 4–15, September 2004.

A Language Feature Analysis

The following sections will provide insight into the concepts of OOP and FP covered in Section 2.1 for Java, C#, Kotlin and Scala. We will also go into more detail about other concepts that have properties that are interesting for our multi-paradigm setting. We will explore how each language has evolved and extended to implement more constructs of the functional paradigm.

A.1 Java

Java is a multi-paradigm programming language created by James Gosling and Patrick Naughton at Sun Microsystems. Java code is compiled to Java bytecode which runs on the Java Virtual Machine (JVM). Any device running a JVM can therefore run compiled Java code. At the time of writing, Java 17 is the latest LTS release.

A.1.1 Type system

Like any of the other MP languages, Java contains two sets of types in their type system, namely primitive types and reference types [74]. The primitive types are byte, short, int, long, char, float, and double. Primitive values do not share state with other primitive values. Reference types are any class or interface type, array types (e.g., int[]) or type variables (used for type parameterisation). Available reference types are shown in Listing 2. There is also a special *null type*, but it is not possible to declare a variable of the null type or cast to null [74].

```
class Scoreboard { // Class ’Scoreboard’ reference type
    int[] scores;  // Int array reference type
    String name;   // String reference type
}
interface IPoint { ... } // Interface ’IPoint’ reference type
record Point(int x, int y) { ... } // Record class ’Point’ reference type (since Java 17)
enum Direction { ... } // Enum class ’Direction’ reference type
interface Box<T super java.lang.Number> { ... } // ’T’ is a type variable reference type
```

Listing 2: Java reference types

To indicate if a variable is immutable, we can use the `final` modifier. A common area of errors is assuming reference types containing the *final* modifier cannot be altered. In Java, variables containing a reference type which are marked as *final* must always contain the reference to that object, but the object’s state may be changed with operations on the object. This also applies to arrays, since they are objects.

Although Java’s philosophy being *names matter* [75], it has adopted type inference as shown in Listing 3. Java 1.7 saw the introduction of the diamond operator (`<>`), which adds type inference and reduces the verbosity in assignments when using generics. Java 10 introduced the `var` keyword, which infers the type of a variable at declaration.

```
// Diamond operator, infers Diesel from reference type in the assignment
Car<Diesel> myCar = new Car<>();
var x = 50; // Infers int type
```

Listing 3: Java type inference

A.1.2 Encapsulation

At the top-level of Java, we find modules. Modules, as the name suggests, increase modularity of software systems and were introduced in Java 9. Modules contain one or more packages and must explicitly state its dependencies. The key motivation of the module approach is strong encapsulation. Packages in Java are containers for a set of Java files. It is not mandatory to create modules nor packages to create an executable Java program.

Besides modules and packages, encapsulation is enforced by access modifiers. There are four types of access modifiers; *private*, *protected*, *public*, and default with no keyword. Access modifiers can be applied to classes, interfaces, enums, records, constructors, variables, methods, or data members.

Since Java 7, an interface's body can contain *public abstract* methods. This requires the implementing class to override the abstract method. Since Java 8, the *public static* and *public default* modifiers were added. Public static methods are not bound to an object, thus can be called using the interface class. Static methods increase the degree of cohesion by putting together related methods in one single place. Since Java 9, we can declare *private* methods and *private static* methods in interfaces.

To restrict what can be a subtype, we can use the `sealed` type introduced in Java 17 [76]. We can define this behaviour to a class, an abstract class, or interface. To allow particular subtype(s), a permits lists can be added to the sealed type as shown in Listing 4.

```
sealed interface Node permits Leaf { ... }
```

Listing 4: Java sealed type

A.1.3 Classes

A class is the blueprint for its objects (i.e. class instances). It contains a name and a body. In this body, we can have zero or more constructors, fields, methods, inner classes, or inner interfaces. The `static` can be combined with access modifiers to indicate that the static member should be applied to a class and not the class instance.

A special type of class that defines a small set of named class instances is the `enum` class. Enums can be declared at top-level, inside a class or as a local declaration. Enums contain a set of constants and optionally some member declarations like fields, methods, classes or interfaces.

Classes are the foundation of inheritance and polymorphism in Java. Every class extends the `java.lang.Object`, which provides some utility methods. Every class supports single-inheritance by extending at most one other class with the `extends` keyword. The direct ancestor of a class is called the superclass, all descendants of a class are called subclasses. A class can implement multiple interfaces with the `implements` keyword.

Java supports polymorphism in several ways. *Method overloading* is supported with the declaration of different types of parameters and return types for methods with the same name. *Method overriding* allows a subclass to override the implementation of a method in one of its ancestor classes. Overriding is not possible for final methods, constructors or static methods in the ancestor class.

We can also instantiate an anonymous class based on an interface. As shown in Listing 5, the name of the interface is used as the class declaration and the methods of the interface are implemented.

```
interface Runnable {
    public void run();
}

class Example {
    public void show() {
        Runnable r = new Runnable() {
            public void run() {
                // ...
            }
        }
    }
}
```

Listing 5: Java anonymous class

A.1.4 Objects

An object is a class instance or an array [74]. The class `Object` is the superclass of all other classes within Java. Reference values, as covered in Type system, are pointers to these objects. Objects contain zero or more methods which interact and possibly modify the state within and outside of the object. Classes that have the `abstract` modifier cannot be instantiated, but its non-abstract subclasses can be instantiated.

A.1.5 Interfaces

An interface is an abstract type that can contain methods and constant values. Classes can implement multiple interfaces. Interfaces can also extend another interface. An interface can be described as a contract that a class should conform to. As we have seen in the Encapsulation section, interface methods can be *default*, *static* or *abstract*. Default methods provide a standard implementation with the interface but allows the implementing class to override it. Abstract methods must be overridden by the implementing class.

A.1.6 Type parameterisation

Java supports the use of type parameterisation in several ways. The first becomes apparent in the creation of generic classes and interfaces as shown in Listing 6.

```
class Stack<T> {}
interface Box<T> {}
```

Listing 6: Java generic types

When it is unknown or unnecessary to know the type of the object that is used in a method, but we want to be able to apply the method on multiple types, we can use generic methods. This is shown in Listing 7, where we transform an array of type T to a collection of type T. Within the `<>` in the method signature, multiple parameter types can also be expressed.

```
public <T> List<T> fromArrayToList(T[] a) {
    return Arrays.stream(a).collect(Collectors.toList());
}
```

Listing 7: Java generic method

Assume we have a collection that should contain shapes but we do not know which shape. In this case we can use `Collection<? extends Shape>`, where the question mark is the wildcard and the `extends Shape` defines the *upper bounds*. We could also define the lower bounds with `super Shape`.

A.1.7 Records

Records were introduced in Java 16 to replace classes that were solely used for representing data. Records are immutable and non-extendable. They are similar to record classes in C#, case classes in Scala and data classes in Kotlin. While records reduce the amount of boilerplate code that is required for a class in Java, this boilerplate code is still added by the compiler under the hood. Records are especially useful for traversal of trees, where the nodes are records, as data transfer objects, or as messages in, for example, Kafka [76].

A.1.8 Functions

In Java 8, lambda expressions were introduced to be the core of functional programming. A lambda expression can have zero, one or more parameters. Lambda expressions use *functional interfaces* under the hood and many functional interfaces are included in the Java standard library. A functional interface contains only a single abstract method (SAM), and optionally one or more default methods or static methods. Therefore, you can omit the name of the method when you implement it. A lambda expression is therefore the implementation of an interface in the form of an anonymous class. Common standard library SAM interfaces include *Function*, *Supplier* and *Consumer*. Since we can use interfaces as parameter types in Java, we can create higher order functions in Java. We show lambda expressions and higher order functions in Listing 8.

```java
// Functional interface             // lambda expression
Function<Integer, Integer> square = (a) -> a * a

// Higher-order function
BiFunction<Function<Integer, Integer>, Integer, Integer> calculate = (f, a) -> f.apply(a);

int result = calculate.apply(square, 4); // 16
```

Listing 8: Java functional programming

A.1.9 Streams

The Streams API, introduced in Java 8 is a feature that allows for intermediate operations on a Collection or Array. The Streams API is similar to LINQ in C#. It defines a pipeline of operations, where elements are computed on demand as shown in Listing 9. A stream will only be executed when it is called by a terminal operation (e.g., *forEach*, *collect*, *match*, *count*, or *reduce*). Streams are designed to be used with lambda expressions, they allow for lazy access and they are parallelisable.

```java
Integer[] scores = new Integer[]{ 97, 92, 81, 60 };

// Method-based syntax
Stream<Integer> scoreStream = Stream.of(scores)
    .filter(i -> i > 80); // Only allow values that are bigger than 80

// The query is only executed when use a terminal operation
scoreStream.forEach(System.out::println) // Lambda function reference (::)
```

Listing 9: Java Streams

A.1.10 Pattern Matching

Matching patterns in Java was a long awaited feature that other languages implemented much sooner. Java is currently developing more complex pattern matching features, but most are only available as *preview features*[17] in Java 17. For now, we have type matching with some extra features within if and switch statements as shown in Listing 10.

Java 14 introduces the concept of *switch expressions*, where a matched switch case can directly yield a value to assign to a variable. These expressions must be exhaustive and every case must yield a value. Another introduction by Java 14 was the inclusion of the *arrow label* `->` which, when the case matches, will only execute the expression or statement after the arrow, there is no fall through to the next cases.

```
// Type check pattern with the guarded pattern
if(user instanceof User u && u.getAge() >= 18) { ... }

// Switch expression
Boolean workday = switch(day) {
        case MON, TUE, WED, THUR, FRI -> true; // with lambda expression
        case SAT, SUN: yield false; // with yield
};

// PREVIEW FEATURE: Switch expression with type matching
String formatter = switch(o) {
    case Integer i -> String.format("int %d", i);
    case Long l    -> String.format("long %d", l);
    case Object o  -> String.format("Object %s", o.toString());
}
```

Listing 10: Java pattern matching

A.1.11 Constructs overview

OOP Constructs	FP constructs	MP constructs	Other constructs
Variables Classes Objects Interfaces Type parameterisation Records	Functions Pattern matching Lazy evaluation	Functional interfaces	Streams

Table 20: Java constructs overview

A.2 C#

C#is a multi-paradigm programming language that was designed by Anders Hejlsberg at Microsoft [27]. At the time of writing, the LTS version is C#10. C#primarily used with the .NET framework, a software framework for developing cross-platform applications.

[17] https://openjdk.org/jeps/406

A.2.1 Type system

There are two types in C#: value types and reference types. Value types directly contain their data, variables of reference types store references to their data (i.e. objects). Simple types, enums, structs, and tuples are value types. Classes, interfaces, records, arrays, delegates, and dynamic types are reference types [28].

Reference type variables can reference the same object. This is not possible for value type variables, except for `ref` and `out` parameter variables. `ref`, `in` and `out` are used to state that the parameter passed, respectively, *may*, *cannot* or *must* be modified by the method.

A.2.2 Encapsulation

To encapsulate what can be accessed in which part of code, C#employs multiple techniques. Namespaces are used to organise the types, like classes, structs and records [27].

Depending on the accessibility of the method or property, code outside of a class or struct can be accessed. These accessibility modifiers are `public`, `protected`, `internal`, `protected internal`, `private` and `private protected`. By default, `private` is used.

A.2.3 Classes

Only classes support inheritance. When a class derives from another class (i.e. the *base class*, it will automatically contain all public, protected, and internal members of the base class except its constructors and finalizers. A derived class can only have one direct base class, but the inheritance is transitive.

```
interface ISampleInterface<T> {
    void SampleMethod(T obj);
}

abstract class SampleClass : ISampleInterface<int> {
    public virtual void SampleMethod(int i) { ... }
}

class DerivedClass : SampleClass {
    public override void SampleMethod(int i) { ... }
}
```

Listing 11: C#Inheritance

In Listing 11, we see a derived class that inherits from `SampleClass`. Only when a base class declares a member (except for fields) as `virtual`, a derived class can `override` the member with a custom implementation. When a base class declares a member as `abstract`, any non-abstract class that directly inherits from it must override the member. When a base class declares a member without any special keywords, the derived class can use the `new` keyword to hide the base class member. Classes can also prevent other classes from inheriting from it, or from any of its members, by declaring itself or the member as `sealed`.

Multiple classes can be derived from one base class. E.g., the base class `Shape` has the derived classes `Rectangle`, `Circle`, `Triangle`. At run-time, the base class can be used in method parameters, collections or arrays. All classes that inherit from this base class can be used interchangeably in those situations.

A.2.4 Records

Since C#9, you can declare `record` types and since C#10 also `record struct` to clarify a value type & `record class` to clarify a reference type. Records are primarily used for storing values, with minimal associated behaviour.

A.2.5 Delegates

Delegates are user-defined types of the form `delegate int D(...)`. They represent references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to function types provided by functional languages and the concept of function pointers, but delegates are object-oriented and type-safe. A delegate has an invocation list, which is a list of methods that the delegate represents and that are executed when the delegate is invoked (inherited from `System.MulticastDelegate`).

A.2.6 Interfaces

Interfaces describe a set of functionality that a non-abstract class or struct should implement. Interfaces cannot be instantiated on themselves. Interface methods are public by default, but its access modifier can be changed to any of the access modifiers available. If a method is set to private, it must have a default implementation. Unlike Java, C#interfaces can contain properties.

A.2.7 Functions

In C#, lambda expressions are anonymous functions. With the lambda operator =>, the lambda parameter list is separated from its body. Lambda expressions come in two forms as shown in Listing 12.

```
// Expression lambda
int MultiplyByFive(int i) => i * 5;

// Statement lambda
void Log(string s) => {
    // ... Some write operation to log files ...
    Console.WriteLine(s); // Debug log
}
```

LISTING 12: C#Types of lambda expressions

In the C#type system, lambda expressions have no type definition as the system does not have an intrinsic concept of lambda expressions. We speak of an ‘informal’ type that refers to the delegate type or Expression type to which the lambda expression is converted. From C#10, a lambda expression may have a natural type where the compiler may infer the delegate type from the lambda expression. We can define these delegate types as shown in Listing 13. We can also use built-in C#delegates. If the lambda does not return a value, the built-in `Action<T1, Tn>` delegate can be used. If the lambda returns a value, the built-in `Func<T1, Tn, TResult>` delegate type can be used. Expression lambdas can also be converted to the expression trees, so they can be used in LINQ queries, which we will cover in Section A.2.10. Expression lambdas can also be used in an asynchronous context as an event handler expression.

```
delegate int IntDelegate(int value); // define delegate
```

```

IntDelegate intDelegate = MultiplyByFive; // Typed delegate
Func<int, int> genericDelegate = MultiplyByFive; // Generic delegate with return value
Action<string> actionDelegate = Log; // Generic delegate without return value
```

Listing 13: C#Delegate type variations

Beginning with C#9.0, you can apply the static modifier to a lambda expression to prevent unintentional capture of local variables or instance state by the lambda. A static lambda can't capture local variables or instance state from enclosing scopes, but may reference static members and constant definitions.

Since functions can be referenced in code with delegates, we can define higher order functions. Such a function is shown in Listing 14.

```
int DoMath(Func<int, int, int> f, int a, int b) => f(a, b);

Func<int, int, int> multiply = (a, b) => a * b; // Define math operation
int result = DoMath(multiply, 4, 5); // Insert expression as parameter
```

Listing 14: C#Higher Order Functions

Besides lambda expressions, local functions can be used. Local functions are defined at compile time. They can be referenced from any code location where it is in scope. Recursive algorithms are easier to create using local functions, but the use of local functions or lambda expressions is still largely based on personal preference. Lambda expressions are objects that are declared and assigned at run-time. In order for a lambda expression to be used, it needs to be definitely assigned, which is the `Action` or `Func` variable that it will be assigned to must be declared and the lambda expression assigned to it.

Depending on their usage, local functions can avoid heap allocations that are always necessary for lambda expressions. If a local function is never converted to a delegate (conversion only happens when local function is used as a delegate) and none of its variables captured by the local function are captured by other lambdas or local functions that are converted to delegates, the compiler can avoid heap allocations.

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2 ? 1 : number * nthFactorial(number - 1);
}
```

Listing 15: C#Local Functions

A.2.8 Pattern Matching

Pattern matching was introduced in C#7.0. Since then, every major version supports new types of pattern matching [77]. In C#7, the *declaration pattern*, *constant pattern* and *var pattern* were introduced (see Listing 16, 17 & 18 respectively). C#8 brought the *property pattern* and *positional pattern* (Listing 21 & 22). C#9 completes our list of listings with the *type pattern*, *relational pattern* and *logical pattern* (Listing 19, 20 & 23). The property, positional and logical patterns are recursive, meaning they can contain nested patterns.

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower());  // output: hello, world!
}
```

Listing 16: C#Declaration Pattern

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount switch
{
    1 => 12.0m,
    2 => 20.0m,
    0 => 0.0m,
    _ => throw new ArgumentException("Not supported number of visitors"),
};
if (input is null) {} // null check
```

Listing 17: C#Constant Pattern

```
if (shape is Rectangle { Length: var length } rect && length % 3 == 0)
{ /* This shape is a Rectangle type with a 'length' variable which is a multiple of 3 */ }
```

Listing 18: C#Var Pattern

```
public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
};
```

Listing 19: C#Type Pattern

```
static string IsAcceptableWeight(double kg) => kg switch
{
    < 5.0 => "Too light",
    > 10.0 => "Too heavy",
    double.NaN => "Unknown",
    _ => "Acceptable",
};
```

Listing 20: C#Relational Pattern

```
static bool IsConferenceDay(DateTime date) => date is
{
    Year: 2020,
    Month: 5,
    Day: 19 or 20 or 21
};
```

Listing 21: C#Property Pattern

```
if (rectangle is (20, _) rect)
{ // ...
}
```

Listing 22: C#Positional Pattern

```
if (input is not null) {} // Since C# 9.0: 'not' keyword possible
if (input is not (float or double)) {} // Parenthesized logical pattern
```

Listing 23: C#Logical Pattern

A.2.9 Lazy Evaluation

It means that an object's creation is postponed until it is first used. Lazy initialization is primarily used to improve performance, avoid wasteful computation, and reduce program memory requirements. `Lazy<T>` is thread-safe and provides a consistent exception propagation policy.

```
Lazy<Document> document = new Lazy<Document>(() => IO.ReadLargeDocument());

// ...Do other operations...

// Initializes the Document object after Value attribute is accessed for the first time.
string docInfo = document.Value.metadata;
```

Listing 24: C#Lazy Initialization

A.2.10 Language-Integrated Query (LINQ)

LINQ is a collective name for a set of technologies based on integration of query capabilities directly into C#. With LINQ, a query is a first-class language construct. Query expressions can be used to query and transform LINQ-enabled data sources (e.g., XML documents or any object with interface type `IEnumerable<T>`). LINQ queries are only evaluated when a terminal method is used (e.g., ToList(), Max(), First(), Count(), etc.) which have a list, object or an integer as a return type.

Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. `IEnumerable<T>` queries are compiled to delegates. `IQueryable` and `IQueryable<T>` queries are compiled to expression trees.

```
int[] scores = { 97, 92, 81, 60 };

// Query syntax
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Method-based syntax
IEnumerable<int> scoreMethodQuery = scores.Where(s => s > 80);

// The query is only executed when use a terminating method
foreach (int i in scoreQuery)
{ // ...
}
```

Listing 25: C#LINQ

A.2.11 Partials

The partial modifier can be added to a class, interface, struct or method to split its definition over multiple source files. These partials are combined when the source code is compiled. Once a part of a class, interface, struct or method is declared *partial*, all other parts must also explicitly declare itself *partial*. Partial methods are only allowed to return void.

```
public partial class PartialClass { //...
}

public partial class PartialClass { //...
}
```

Listing 26: C#Partial Classes

A.2.12 Extension methods

Extension methods in C#allow the developer to extend the functionality of a type without creating a derived type. These methods are defined as static methods with at least one parameter prefixed with this to indicate the type that should get the extension. Extension methods are very useful with LINQ. When creating extension methods for collections, it creates a cleaner LINQ query.

A.2.13 Constructs overview

OOP Constructs	FP constructs	MP constructs	Other constructs
Variables	Functions	LINQ	Tuples
Classes	Pattern matching	Delegates	Partials
Interfaces	Lazy evaluation		Extension methods
Type parameterisation			
Records			
Enums			

Table 21: C#constructs overview

A.3 Kotlin

Kotlin is a statically typed, general-purpose, multi-paradigm programming language designed by JetBrains. Kotlin is multiplatform, it can run on the JVM, natively with a LLVM-based backend, and transpiled to JavaScript. At the time of writing, Kotlin's latest version is 1.7.10.

A.3.1 Type system

The language operates with values or object which have types. Empty values are represented by the null object. Type safety is verified statically, at compile time, for the majority of Kotlin types. Null safety is guaranteed with nullable types (i.e. `T?`) and non-nullable types (i.e. `T`). The unified supertype of all types in Kotlin is `kotlin.Any?` and the unified subtype type is `kotlin.Nothing`. The superclass `Any` contains the methods `equals()`, `hashCode()` and `toString()`.

There are several built-in types that Kotlin supports. It supports basic types (i.e., Byte, Short, Int, Long, Float, Double, Boolean, Char and String), parameterised type Array(T), classifier types (i.e., classes, interfaces or objects), and function types.

Types can be aliased with the `typealias` keyword. It decreases the verbosity of types as it can shorter names or create a new one instead.

A.3.2 Encapsulation

Encapsulation in Kotlin is primarily controlled by visibility modifiers. These modifiers apply to classes, objects, interfaces, constructors, functions, and properties including their setters, while getters of the property always have the same visibility as the property.

All elements of Kotlin are contained in packages. The visibility modifiers are `private`, `protected`, `internal`, and `public`. *Public* is the default visibility. Private members are only visible within the same class, and all its members. Protected is similar to the private modifier, but now all members are visible within subclasses also. Internal means that visibility is limited to this module (i.e. compilation unit) only. Public means that it is visible for everyone.

A.3.3 Classes

As shown in Listing 27, *Classes* are declared with a name, header (optionally with type parameters), the primary constructor, and a class body. The header and body are optional. While the primary constructor will be part of the class header, a class can also contain secondary constructors which have to delegate back to the primary constructor. A primary constructor cannot contain any code. Initialisation code can be defined within *initializer blocks* with the `init` keyword.

Classes can also contain nested classes or interfaces which will be bound to the class. Nested classes can also be marked as `inner`, which enables access to its outer class and binds it to the object (i.e. class instance) of the outer class.

```kotlin
class Empty // Class without parameters and body

class Person(val name: String) { // Primary constructor in header
    val children: MutableList<Person> = mutableListOf()

    // Secondary constructor that delegates to primary constructor with 'this'
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Listing 27: Kotlin Classes

By default, Kotlin classes cannot be inherited from. To enable this, an `open` modifier must be added to the class header. Just like classes, its members can only be overridden if it is marked with the `open` modifier. A class (or interface) can also be `sealed`, which makes it abstract and only extendable within the same package and module.

A special type of class in Kotlin is the *enum class*. It can be used for the implementation of type-safe enums, where each enum constant is an object. Enum constants can declare their own anonymous classes with methods, and can therefore also implement interfaces. Listing 28 shows the declaration of an enum class.

```kotlin
enum class IntArithmetics : BinaryOperator<Int>, IntBinaryOperator {
    PLUS { override fun apply(t: Int, u: Int): Int = t + u },
    TIMES { override fun apply(t: Int, u: Int): Int = t * u };

    override fun applyAsInt(t: Int, u: Int) = apply(t, u)
}
```

Listing 28: Kotlin Enum class

A.3.4 Objects

There are two types of objects within Kotlin. The first, *object expressions* shown in Listing 29, creates an instance of an anonymous class. These objects are useful for one-time use. Anonymous objects can be created from scratch, inherit from existing classes, or implement interfaces. Object expressions are executed immediately when they are used. The second, *object declarations* shown in Listing 30, are a class using the Singleton pattern. Companion objects, shown in Listing 31, are bound to a class and apply static behaviour to a class while being able to access a class its internals.

```
val helloWorld = object {
    val hello = "Hello"
    val world = "World"
    // object expressions extend Any, so override is required on toString()
    override fun toString() = "$hello $world"
}
```

Listing 29: Kotlin Object Expression

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) { ... }
}
```

Listing 30: Kotlin Object Declaration

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
val instance = MyClass.create()
```

Listing 31: Kotlin Companion Object

A.3.5 Type parameterisation

Parameterised types are supported in Kotlin in several ways. Classes can contain a generic type parameters as shown in Listing 32. It is also possible for functions to have type parameters, which are placed before the name of the function as shown in Listing 33.

Variance, covariance and contravariance exists in Kotlin just like it does in C#and Scala. Kotlin checks for valid generic type use at compile-time, instead of run-time which is done in C#. Therefore, it does not have to implement all the wildcards that are part of Java (e.g., `? extends E` & `? super E`). The `out` modifier can be used as an indicator of covariance. The `in` modifier can be used as an indicator of contravariance, meaning it can only be consumed and never produced.

```
// Class with generic type parameter T, which is a variant.
```

```
class Box<T> { ... }

// Covariant, T is only returned (produced) from members of Source<T>
interface Source<out T> { //
    fun next(): T
}

// Contravariant, T is only consumed and never produced
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}
```

Listing 32: Kotlin Variance

The type within a generic function can be constrained with an upper bound, as shown in Listing 33. Generic types can be type-checked for multiple extensions with the `where` clause within a generic function.

```
// Check if T is a subtype of Comparable<T>
fun <T : Comparable<T>> sort(list: List<T>) {  ... }

// Check if T is a subtype of both CharSequence and Comparable<T>
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

Listing 33: Kotlin Generic Function with Where clause

A.3.6 Interfaces

Interfaces in Kotlin are different from abstract classes as they cannot store state. Interfaces are very useful as classes can inherit from one or more interfaces. Interfaces can contain abstract methods but they can also have method implementations. Interfaces can also contain properties, but these need to be abstract or provide accessor implementations. Listing 34 demonstrates the possibilities of an interface in Kotlin.

```
interface Foo {
    val bar: Int // abstract
    val fooImpl: String
        get() = "foo"

    fun foo() {
        print(bar)
    }
}
```

Listing 34: Kotlin Interface

A.3.7 Data classes

When a class its purpose is solely to hold data, we can use *data classes* as shown in Listing 35. Data classes must contain at least one parameter, which must be marked with `val` or `var`, and the class cannot be abstract, open, sealed, or nested. A data class can use *destructuring declarations* in the form `componentN()` that return the value at the `N`th place.

Destructuring declarations are not limited to data classes, but can be implemented for all other classes as well. The copy() function is used to copy values in the class instance, allowing you to alter some of the properties while leaving the rest unchanged.

```
data class User(val name: String, val age: Int)

val user = User("John", 20)
val name = user.component1() // "John"
val userCopy = user.copy(age = 21) // Copy object but change the age
```

LISTING 35: Kotlin Data class

A.3.8 Functions

As shown in Listing 36, functions can contain parameters, which must be explicitly typed. Function parameters support default values. Functions return Unit if no return type is specified. The last function parameter can be prefixed with the vararg modifier, which allows for a variable number of arguments to be inserted into the function.

```
// Simple function with default value for x
fun double(x: Int = 0): Int {
    return x * 2
}

val ten = double(x = 5) // Function call with named parameter

fun triple(x: Int): Int = x * 3 // Single expression function

// Infix function
infix fun Int.equals(x: Int): Int {
    return this == x
}
val isEqual = 5 equals 5 // true

// Operator overloading a unary operation in data class Point
operator fun Point.unaryMinus() = Point(-x, -y)
val point = -Point(10, 20) // "Point(x=-10, y=-20)"
```

LISTING 36: Kotlin Functions

Functions are first-class citizens, which make it possible to create higher order functions like shown in Listing 37. This listing also shows the definition of a lambda expression, which is a function literal. Function literals are functions that are not declared but immediately passed as an expression.

```
val lambdaExpression: (Int) -> Int = { x: Int -> x * 2 }

// Lambda argument types are optional, they are inferred by the compiler
val higherOrderLambdaExpression ((Int) -> Int, Int) -> Int = { f, x -> f(x) }

fun higherOrderFunction(lambdaExpression: (Int) -> Int, x: Int): Int {
    return lambdaExpression(x)
}
```

LISTING 37: Kotlin Lambdas & Higher Order Functions

A.3.9 Pattern Matching

The extend to with Kotlin has adopted pattern matching is less than C#or Scala. One could argue that pattern matching is not *really* integrated into Kotlin. We can match on some patterns with a *when expression*. These expressions can match on expression (i.e. value), range or type as shown in Listing 38.

The `when` expression can also be used without a subject in its header. When we analyse this code from Kotlin bytecode decompiled to Java code, we see that the when expression is transformed to a set of Java if-elseif statements. If `when` is used with a subject, such as `x` in the listing, the bytecode shows a transformation to a Java switch statement.

```
when(x) {
    1           -> println("Match with expression (value)")
    in 1..10    -> println("Match in range")
    is Int      -> println("Match by type")
    else        -> println("Match everything else")
}
```

Listing 38: Kotlin Pattern Matching

A.3.10 Currying

With the addition of first-class functions, currying is also possible within Kotlin as shown in Listing 39.

```
val add: (Int) -> (Int) -> Int = { x -> { y -> x + y } }
val result = add(5)(10) // 15
```

Listing 39: Kotlin Currying

A.3.11 Delegated properties

Some kind of properties are easier to implement once to add to a library for later use than to implement them manually every time they are needed. In Kotlin, these are called delegated properties with syntax `val/var <property name>: <Type> by <expression>`. It is used for *lazy* properties which are computed only on first access, *observable* properties which notifies listeners about changes to this property, and storing properties in a *map* instead of a separate field for each property.

While Kotlin does not have native lazy evaluation, it is part of Kotlin's Standard Library as a delegated property. It is a function that takes a lambda and returns an instance of `Lazy<T>`, which can serve as a delegate for implementing a lazy property.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
} // Lazy<String>

fun main() {
    println(lazyValue) // Property is evaluated on first access, prints: computed! Hello
    println(lazyValue) // Only prints: Hello
}
```

Listing 40: Kotlin Lazy Evaluation

A.3.12 Constructs overview

OOP Constructs	FP constructs	MP constructs	Other constructs
Variables Classes Objects Interfaces Type parameterisation	Functions Currying Pattern matching Lazy evaluation	Data classes	Sequences Extension functions Delegated properties Null safety

Table 22: Kotlin constructs overview

A.4 Scala

Scala is a multi-paradigm language that is compiled to run on the JVM. At the time of writing, Scala 3 is the latest version. Scala interoperates with Java, all the extra features that Scala provides are compiled as close as possible to Java [78].

A.4.1 Type system

Scala is statically typed and enforces at compile-time that abstractions are used in a safe and coherent manner [79].

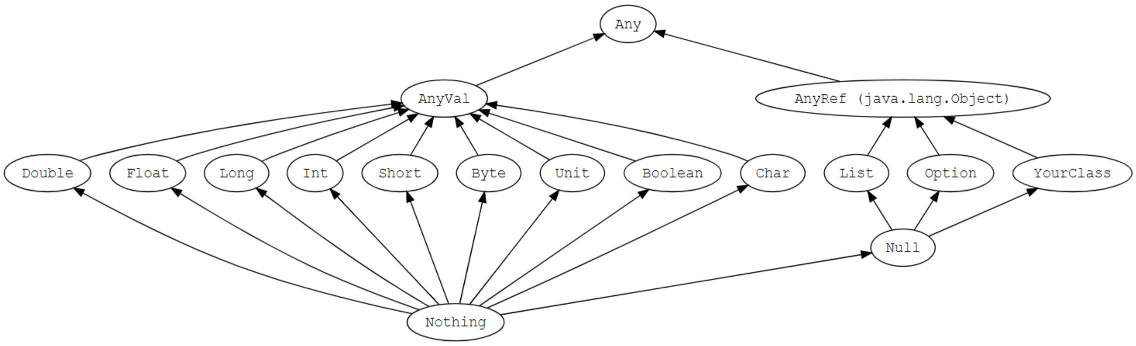


Figure 33: Scala type hierarchy [78]

Figure 33 depicts the type system of Scala. At its core, it contains the supertype *Any*, which provides the universal methods *equals*, *hashCode* and *toString*. The *Any* class has two subclasses, *AnyVal* and *AnyRef*.

AnyVal contains nine predefined non-nullable value types. While most are common within programming language type systems, the *Unit* type is one unique to Scala. It carries no meaning information within its value. There is one instance of type *Unit* written as `()`. The *Unit* type is very useful as a return type since all functions must return something. *AnyVal* subclasses can be mutable with the `var` keyword or immutable with the `val` keyword.

AnyRef represents reference types. Every non-value type is defined as a subclass of *AnyRef*. Due to Scala being part of the JVM, *AnyRef* is the Scala equivalent to `java.lang.Object`.

```
val list: List[Any] = List(
  "a string",
  732,  // an integer
  'c',  // a character
  true, // a boolean value
  () => "an anonymous function returning a string"
)
```

Listing 41: Scala types

Due to Scala's type safety, there must always be a type, even if there is not one. Therefore, *Nothing* is a subtype of all types, called the bottom type. There is no value that has type *Nothing*. It is the type of an expression which does not evaluate to a value, or a method that does not return normally.

Null is a subtype of *AnyRef*. It has a single value identified by the keyword literal `null`. *Null* was added for interoperability purposes with other JVM languages and, by convention, should not be used in Scala code [78].

A powerful function of the Scala type system is its ability to infer the type of values that aren't explicitly typed by the developer. In that sense, Scala feels like a dynamically typed language, but the compiler will provide a type to variables and values.

A special type of immutable collection is the *Tuple*, which can contain a fixed number of elements notated as `(V1, V.., V..n)`. Tuples are especially useful in pattern matching. Just like C#'s Tuple, the *Tuple* in Scala has an alternative called *case classes* that many prefer due to an improvement in readability.

A.4.2 Encapsulation

Scala code is encapsulated in *packages* that contain *classes*. The entry point of a program is indicated with a `@main` annotation followed by a method definition in Scala 3. To determine which code can be reached, the access modifiers `private`, `protected` and *public* are used. Classes and its members are *public* by default.

A.4.3 Classes

A *Class* is a blueprint for creating objects. They can contain members, where a member can be a method, value, variable, type, object, trait or inner class.

```
@main
def run(): Unit = {
  println(Point2D(2, 3))  // (2, 3)
  println(Point2D(y = 5)) // (64, 5) (named parameter)
  println(Point2D())      // (64, 0)
}

// Constructor using default values
class Point2D(var x: Int = 64, var y: Int = 0) {
  def move(dx: Int, dy: Int): Unit =
    x = x + dx
    y = y + dy

  override def toString: String = s"($x, $y)"
}
```

Listing 42: Scala class

As stated earlier, a class can also be a class member. This is what we call an *inner class*. Opposed to Java-like non-static inner classes which are members of their enclosing class, inner classes in Scala are bound to the outer class instance. Listing 43 shows how the inner classes are bound.

```scala
class Graph {
  class Node {
    // connect another Node that is bound to this Graph instance.
    def connectTo(node: Node): Unit = ...
  }
  def newNode: Node = ... // create new Node
}

val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode

val graph2: Graph = new Graph
val node2: graph2.Node = graph2.newNode
node1.connectTo(node2) // fail: graph1.Node cannot connect to graph2.Node!
```

Listing 43: Scala inner class binding

A.4.4 Objects

Objects in Scala are classes that have exactly one instance. They are used for lazy evaluation when it is referenced. Objects are singleton if they are defined as a top-level value. As a member of a class or as a local value, objects are lazy evaluated and thus only initialised the first time that they are called.

An object with the same name as a class is called a *companion object*. The class is also the object's companion class and each can access each others private members. Companion objects are used for methods and values that are not bound to companion class instances (i.e. the static members). Within companion objects, we can create a factory method with `apply()`.

```scala
import scala.math.{Pi, pow}

class Circle(radius: Double):
  import Circle.* // Import required to use calculateArea method
  def area: Double = calculateArea(radius)

object Circle:
  private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)
```

Listing 44: Scala companion object

In a companion object, we can define a Factory pattern. `apply()` can be used as a Factory method. If this pattern is implemented, the Scala compiler allows developers to create new instances of a class without using the `new` keyword. These class instances can also be dismantled with the `unapply()` method. This method extracts all object values and lists them as a result. This is most often used in pattern matching and partial functions.

A.4.5 Type parameterisation

Within Scala, we can use type parameterisation to create *generic classes*. It is a convention to use the letter A as identifier, but any identifier name is allowed. Listing 45 shows a Stack implementation using the generic type.

```scala
class Stack[A] {
  ... // stack implementation
}
```

```

val stack = new Stack[Int]
stack.push(1)
```

Listing 45: Scala generic class

A powerful addition to generic types is Scala's use of variance annotations for type parameters. There are three types of variances, explained by the following bullets points and Listing 46

- *Invariance*: By default, every generic class is an invariant. An invariant in the context of generic classes means that the type that is specified must be the type that is used. E.g., if we have a container filled with values of type `A`, we can only insert new values of type `A`, not values that extend `A`, or that are the base type of `A`.
- *Covariance*: Given some class `Container[+A]`, then if `A` is a subtype of `B`, `Container[A]` is a subtype of `Container[B]`. This allows us to make very useful and intuitive subtyping relationships using generics.
- *Contravariance*: Given some class `Container[-A]`, then if `A` is a subtype of `B`, `Container[B]` is a subtype of `Container[A]`.

```
class Container[A]   // An invariant class
class Container[+A]  // A covariant class
class Container[-A]  // A contravariant class
```

Listing 46: Scala class variants

A.4.6 Traits

Traits are used to share interfaces and fields between classes with extension. Traits themselves cannot be instantiated. Traits are often used as generic types to assign generic behaviour to a set of classes with abstract methods. Although traits share many similarities with interfaces as can be found in C#, Kotlin and Java, they are inherently different due to the possibility of incorporating *state* within a trait. A trait can also be used to describe *dependencies* of that trait that each extending class should implement. Each dependency can be indicated with a *self-type*, where we define the dependant trait as a field followed by a =>. This requires the class extending this trait A to also extend the dependant trait.

```
trait Iterator[A]:
  def hasNext: Boolean
  def next(): A

class DoubleIterator(to: Double) extends Iterator[Double]:
  private var current = 0
  override def hasNext: Boolean = current < to
  override def next(): Double =
    if hasNext then
      val t = current
      current += 1.0
      t
    else
      0
end DoubleIterator
```

Listing 47: Scala Trait

A special type of *trait* is a *mixin*. This is a trait which extends an abstract class. Therefore, all concrete subclasses of that abstract class can use this mixin. This adds specialised functionality to a subclass using the mixin, while keeping the abstract class pure with core functionality.

Traits are also used for compound types. Imagine we have a function that takes as a parameter an object, but it does not have to know which specific class it belongs to. If we only require functionality that is defined within traits, we can use the trait as the parameter type. When we require compound types (i.e., more than one trait), we can use the `with` keyword to add more traits that the parameter must support.

A.4.7 Methods

As shown in previous listings, methods are defined with the `def` keyword. A method can take zero or more parameters and requires a return type, which can be *Unit* if no value is returned. A method can be nested within another method, making it only callable within that outer method.

Methods can have multiple parameter lists. It can be used to drive type inference when multiple generic types are used within the parameter list as shown in Listing 48. This method also becomes *polymorphic* due to the generic type(s).

```scala
def foldLeft[A, B](as: List[A], b0: B, op: (B, A) => B) = ...
def foldLeftMultiParam[A, B](as: List[A], b0: B)(op: (B, A) => B) = ...

// Compile error, A & B are inferred on 'types' _ + _
def fail = foldLeft[Int, Int](numbers, 0, _ + _)

// Does compile, types are inferred in 1st parameter list
def success = foldLeftMultiParam(numbers, 0)(_ + _)
```

Listing 48: Scala Multi-Parameter Type Inference

Parameter lists are required to specify `implicit` / `using` parameters if there are also non-implicit parameters used for the method. Methods can also be called with fewer number of parameter lists than specified. This is known as partial application [80] and will yield a function taking the missing parameter lists as its arguments.

Just as traits can be used to extend the functionality of a class, extension methods can be used to extend to functionality of a class as shown in Listing 49. It is especially useful when you do not have access to the class you want to extend due to external dependencies.

```scala
extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2
```

Listing 49: Scala Extension Method

A.4.8 Case classes

Case classes are similar to C#'s *Records*. As shown in Listing 50, case classes are used for modelling immutable data. All parameters of case classes are a *public* `val` by default. It is possible, but discouraged, to define `var` parameters. Case classes are compared by structure, not by reference. It is possible to copy values to another case class object with the `copy` method.

```scala
case class TaxReport(name: String, dueDate: String, recipient: String)
val report1 = TaxReport("2022-Q3", "21 Oct 2023", "John Doe")
val report2 = report1.copy(recipient = "Foo Bar")
```

Listing 50: Scala Case Class

A.4.9 Functions

In Scala, functions are a big part of the language. They are first-class citizens of the language. Anonymous functions (i.e. function literals) can be assigned to a variable as shown in Listing 51. Functions can be shortened when the function parameter only appears once in the function body as shown on line 10 in our listing.

Since functions are first-class citizens of Scala, they can be used as a function's input parameters or to return a function from a method. In our listing on line 10 to 13, we show different types of higher order functions.

Although functions look very similar to methods, they do not support default parameter values or named arguments. Leveraging the Eta Expansion in Scala 3, methods are automatically converted to function types when they are used as a function variables, as shown in our higher order function example on line 12 in Listing 51.

```
 1 def powMethod(x: Int): Int = x * x          // Method definition
 2 def higherOrderMethod(x: Int, f: Int => Int): Int = f(x)
 3
 4 val powFunction = (x: Int) => x * x         // Function definition
 5 val higherOrderFunction = (x: Int, f: Int => Int) => f(x)
 6
 7 powMethod(5)      // Call by method
 8 powFunction(5)    // Call by function
 9
10 higherOrderFunction(5, _ * 2) // Call by function w/ (shortened) function as parameter
11 higherOrderFunction(5, powFunction) // Call by function w/ function as parameter
12 higherOrderFunction(5, powMethod) // Call by function w/ method as parameter
13 higherOrderMethod(5, x => x * x) // Call by method w/ anonymous function as parameter
```

Listing 51: Scala Functions

Annotations in Scala are used to associate meta-information with definitions. It can give suggestive information to the compiler to give a warning or fail the compilation if a certain condition is met. E.g., @tailrec ensures that a function is tail-recursive, which requires memory to be kept to a certain constant level.

A.4.10 Pattern matching

Pattern matching in Scala looks very similar to a switch statement in Java. It is a functional construct that allows for a wide range of functionality, matching on enums, (case) classes, tuples and values. Extractor objects allow classes to be matched on value within a pattern matching expression. In Listing 52, we list the patterns.

Pattern matching expressions can also be exhaustive on type matching when the type that is being matched contains a sealed base class or if it is used as part of an assignment statement. The compiler checks if all types extending the sealed base class are included in the cases of the match expression.

```
case class Person(name: String, age: Int)

val a: Any = ???
a match {
  case 25 => ... // Value match
  case (_, 5) => ... // Tuple value match
  case Person => ... // Type match
  case Person(name, age) if age > 18 => ... // Type match with value guard
  case Person(_, age) => ... // Type match and extract values, 'age' accessible
  case _ => ... // Match everything else
}
```

Listing 52: Scala Pattern Matching

A.4.11 Currying

As we have seen in Listing 48, methods can contain multiple parameter lists. We can refer to these lists as "curried" methods. Shown in Listing 53, functions can also be curried and will behave the same as curried methods. As we have mentioned before, this is due to the Eta Expansion converting methods into functions where required.

```
def multiplyMethod(n1: Int)(n2: Int) = n1 * n2
val multiplyCurry = (n1: Int) => (n2: Int) => n1 * n2

multiplyMethod(3)(4)  // 12
multiplyCurry(3)(4)   // 12
```

Listing 53: Scala Currying

A.4.12 For-comprehensions

With the notation `for ( enumerators ) yield e`, for-comprehensions can be expressed in a lightweight way. With this construct, new collections can be created (using the `yield` keyword) from the existing collection that is being iterated over.

```
val students =
  for user <- users if user.age >= 17 && user.age < 30
  yield user.name  // i.e. add this to a list of strings
```

Listing 54: Scala For Comprehensions

A.4.13 Lazy evaluation

By default, Scala is strictly evaluated. Strict evaluation is common in OOP languages while lazy evaluation is common within FP languages. Scala supports lazy evaluation with the `lazy` keyword in front of a variable or expression. In that case, variables and expressions are only evaluated when they are needed. Although it is better for memory management and performance, the developer cannot rely on the execution order.

A.4.14 Constructs overview

OOP Constructs	FP constructs	MP constructs	Other constructs
Variables Classes Companion objects Type parameterisation Enums Traits	Functions Currying Pattern matching Lazy evaluation	For-comprehensions Extractor objects Case classes	Tuples Extension methods

Table 23: Scala constructs overview

B Metamodel Design Iterations

The 'generic representation' was deliberately kept vague in the general architecture of our framework to give freedom in a later stage of this research to create the ideal metamodel. This metamodel is one of the core points of this framework and is created iteratively. This iterative approach was chosen to split the creation and modelling of this metamodel into understandable pieces.

Aggregating, merging, and abstracting from multiple programming language feature models and grammars is no easy feat, and can incur many mistakes in the process. By starting at one point and working to improve, we create a metamodel that fits best for our purpose, with a record of implementation choices and assumptions.

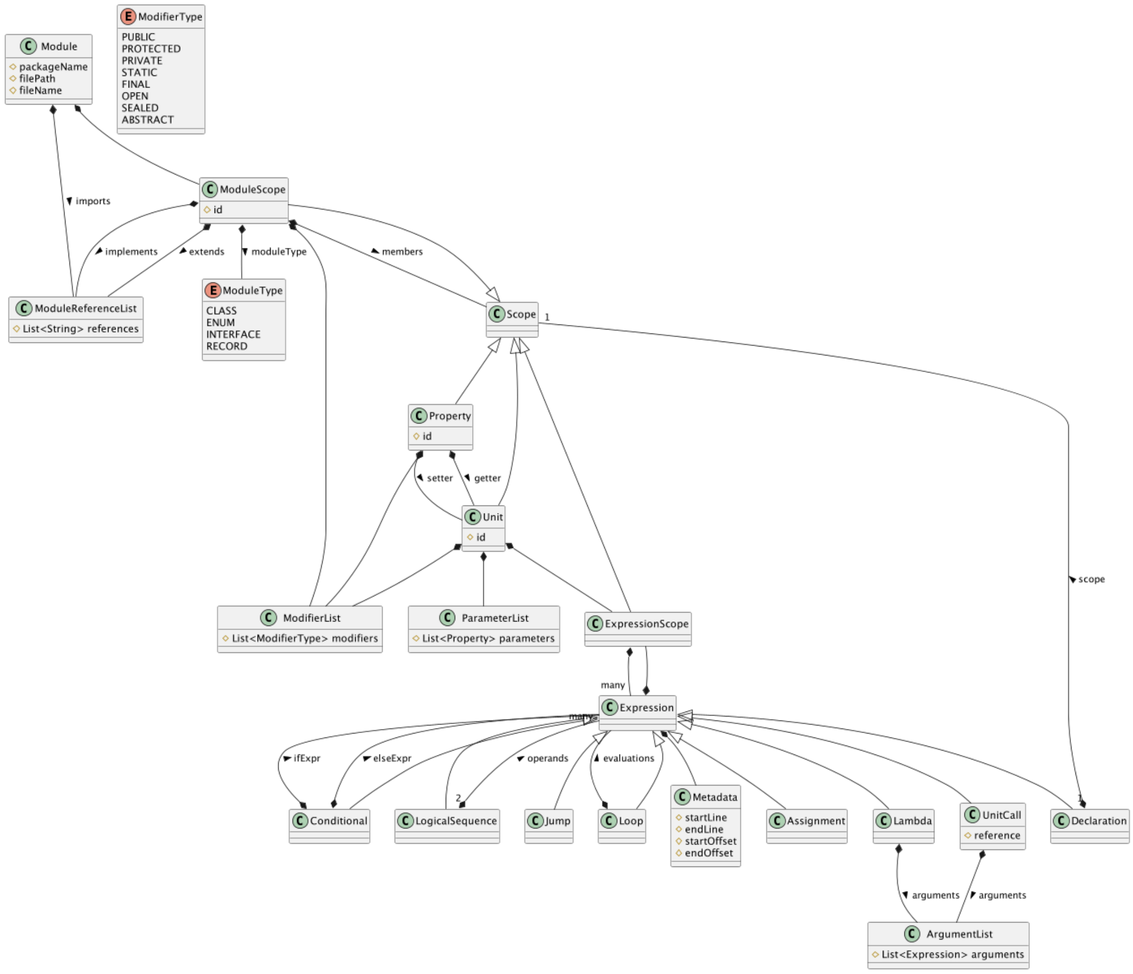


Figure 34: Metamodel v2.1

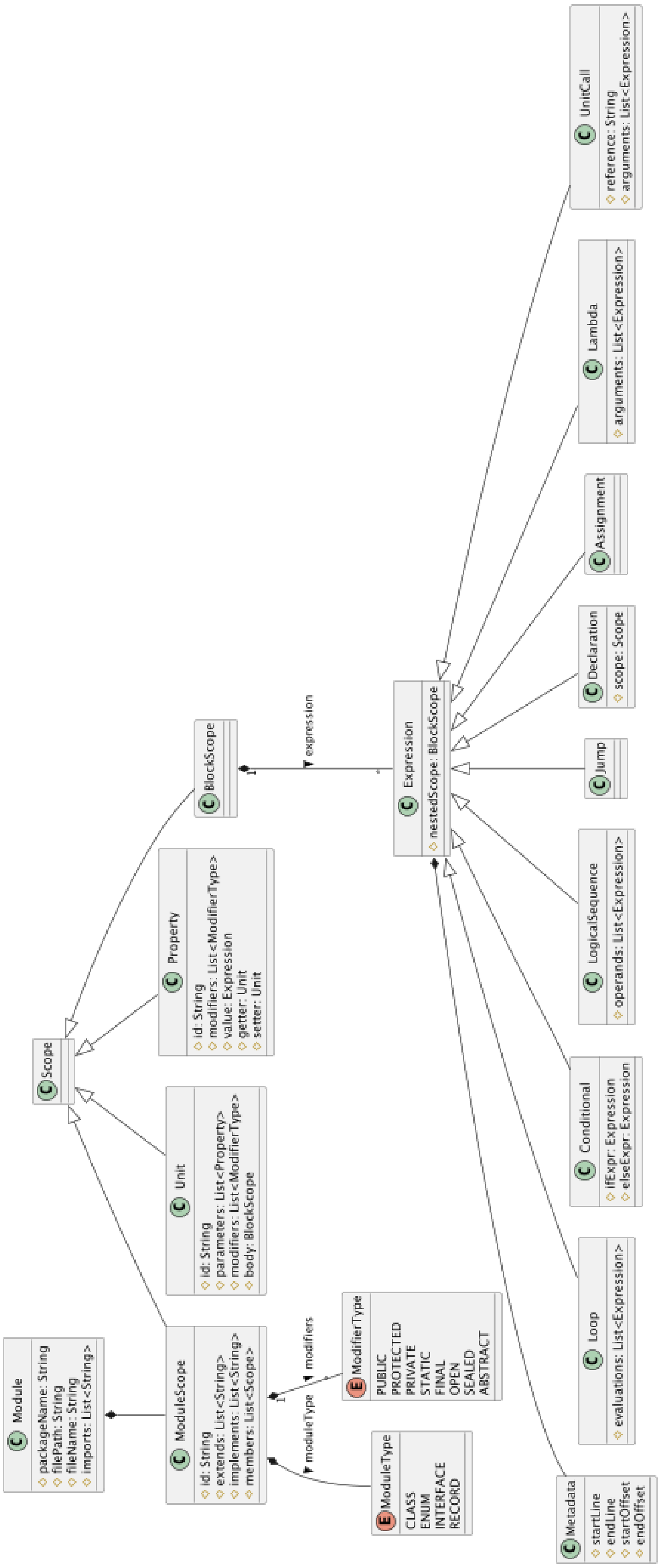


FIGURE 35: Metamodel v2.2

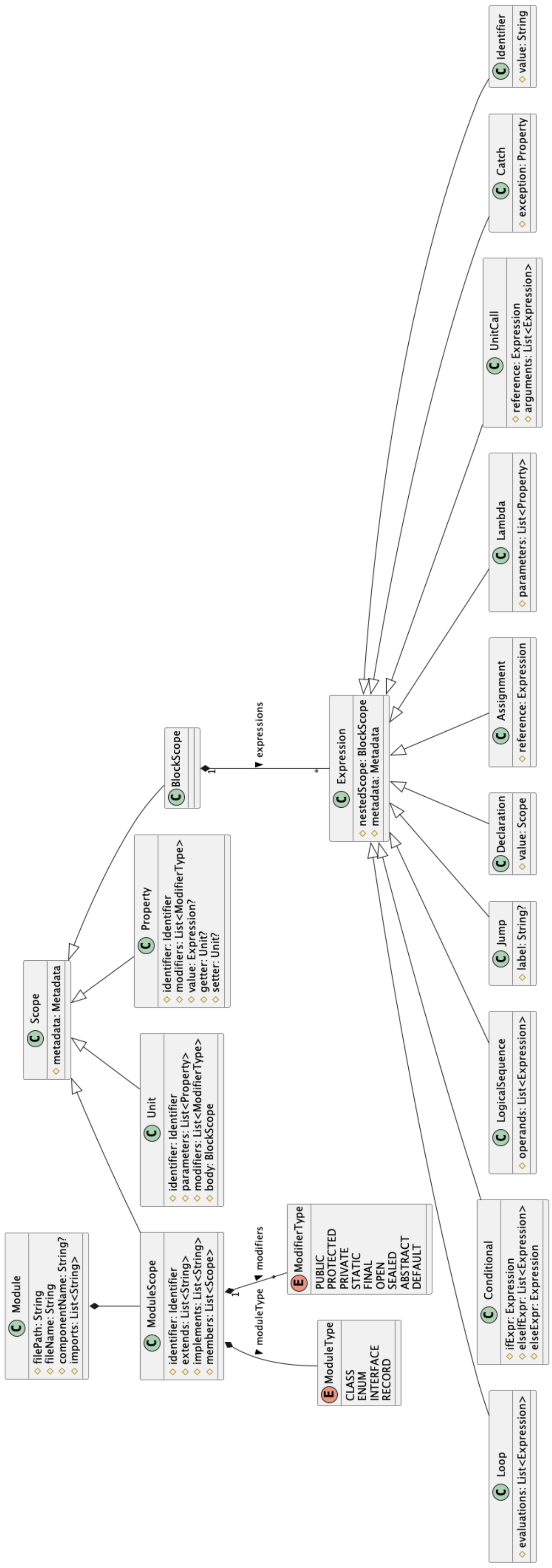


FIGURE 36: Metamodel v2.10

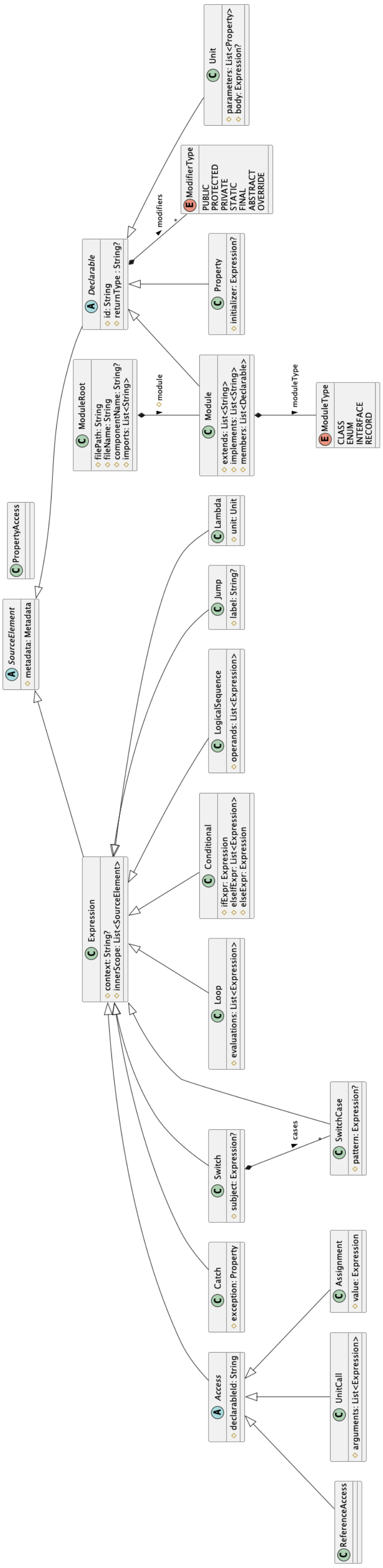


Figure 37: Metamodel v3

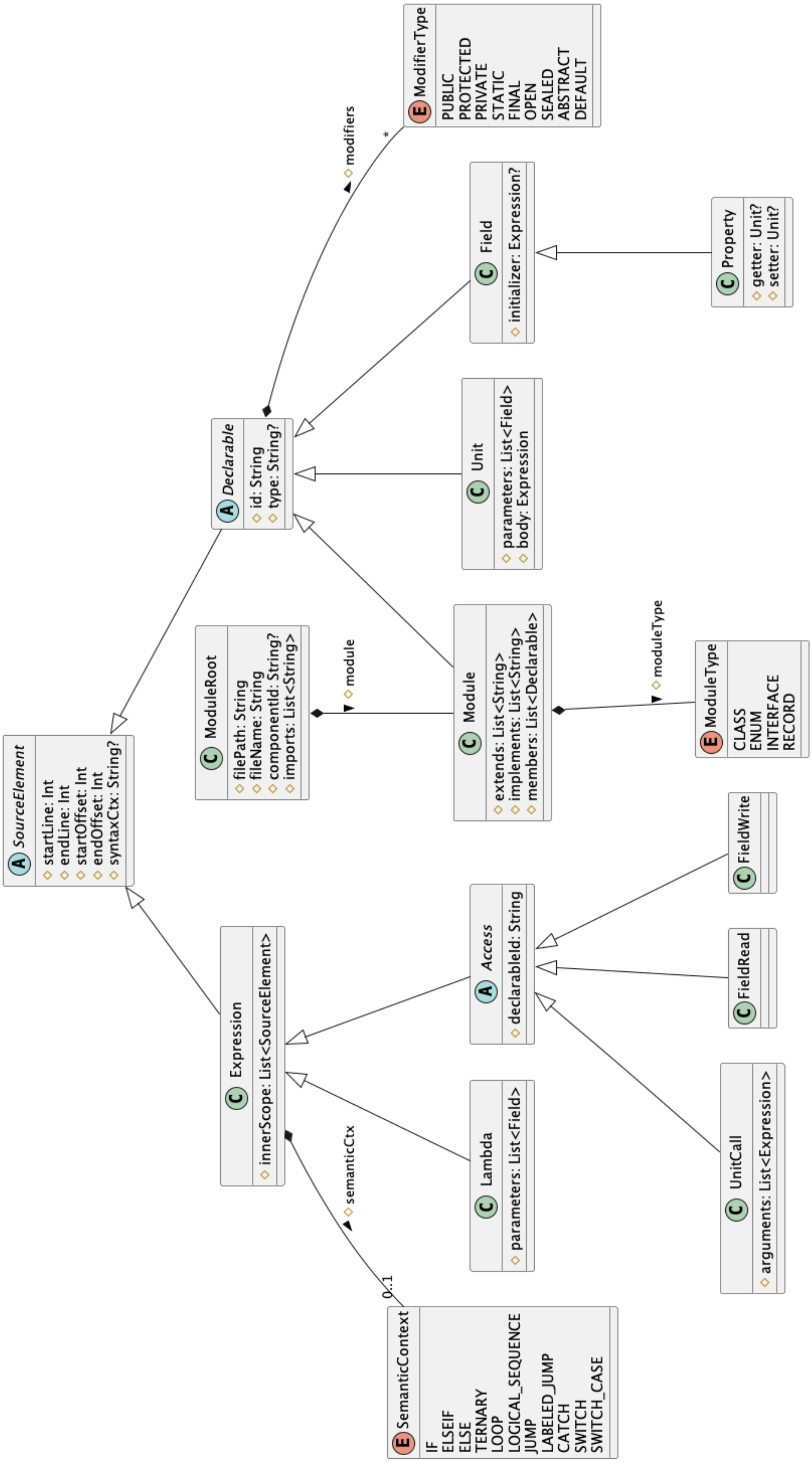


FIGURE 38: Metamodel v4

B.1 Metamodel v1

The first version of the metamodel was created by sticking close to the grammar of Java. Java was chosen for this task because it is very extensive, has much documentation about its grammar and was already familiar to me. Sticking close to the grammar was very useful for understanding how the language of Java was built up, but it did not provide the flexibility that is required for modelling a model that should be able to model the knowns and unknowns.

B.2 Metamodel v2

What assumptions can be made about the metamodel design before we know how to precisely implement it? To answer that question we have to ignore much of what we learned about error checking and type checking, because we can assume that everything that is being put into the generic AST has already compiled.

When we look at the metrics and we compare it to the massive size of language grammars, we can surely conclude that we can strip a lot of information from the ASTs for our implementation of the generic AST.

When looking at the GASTM, we still see a lot of specifics that are required to model every language in the same way, but that model serves a different purpose than ours. It might be used for transforming one language to another, where deletion of semantics can give problems.

Our metamodel needs to be able to give us a way to measure the chosen metrics. This relieves us from some tedious work in analysing specifics about a language. With the metrics in mind, we stripped away much of the structure of grammars of our selected languages and saw a common structure within.

On a high level, we talk about encapsulation of elements, a.k.a. scoping. We stick to terminology found in previous research, mostly by SIG. A project is the whole source code, including its dependencies. Components are internal packages that contain modules. Modules are classes, interfaces, enums or records (and structs?). Modules can contain members, which we modelled as scopes. There are several types of scopes: the state a.k.a the properties, and the units a.k.a the work units that can interact with the state, and inner modules, which provide further encapsulation within modules, and also expression scope that can model language specific semantics not captured within our other models.

We combined fields and properties into one and the same generic model. It is, for every language, a field with a (default) getter and setter. When a field is immutable, the setter is left undefined and so on in terms of behaviour. This getter and setter are special Units bound to the property.

A Unit in our metamodel is a packet of work that can contain a list of parameters. These parameters are just properties (i.e. have a default getter and setter). A Unit also has an internal scope, called the UnitBody. Parameter properties are only accessible within this scope. The unit body contains zero or more Expressions which leads us to the final part of the puzzle.

Within a UnitBody, everything is an expression. We bundle all possible behaviour together as expressions. This means that statements are now expressions that return Void/Unit/Nothing and the expressions we all know are now also modelled as expression. This gives us more freedom to map our languages. To be able to correctly measure some of our code metrics, we created some subclasses for expressions. Conditionals are modelled separately (if, if else, else, when, switch), LogicalSequences (bundles of && expressions), Jumps (continue, break), UnitCalls, Assignments (changes of state), Loop (while, for), Declarations (property, unit, module), Lambda (which is a UnitCall and Unit definition at the same time).

B.3 Metamodel v3

As depicted in Figure 37, we changed the general look and feel of the metamodel. Integrated Switch and SwitchCase into the metamodel. Generally, the structure was improved to make it easier to implement the symbol tree information. But that proved to be impossible with the current semantic information included in the metamodel.

B.4 Metamodel v4

As depicted in Figure 38, we removed a large part of the expression subclasses. The information was not important for the calculation of our metrics. Added Field as a supertype of Property to better describe the parameter lists of Lambda and Unit. We created the Access abstract class with FieldRead, FieldWrite, and UnitCall. It better describes how the program works.

C LAMP Metamodel XSD

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="https://www.utwente.nl/v3"
           xmlns="https://www.utwente.nl/v3"
           elementFormDefault="qualified">

    <xs:element name="ModuleRoot">
        <xs:complexType>
            <xs:sequence>
                <xs:sequence id="imports" minOccurs="0" maxOccurs="unbounded">
                    <xs:element name="Import" type="xs:string"/>
                </xs:sequence>
                <xs:element name="Module" type="Module"/>
            </xs:sequence>
            <xs:attribute name="componentName" type="xs:string"/>
            <xs:attribute name="filePath" type="xs:anyURI"/>
            <xs:attribute name="fileName" type="xs:string"/>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="Metadata">
        <xs:attribute name="startLine" type="xs:positiveInteger"/>
        <xs:attribute name="endLine" type="xs:positiveInteger"/>
        <xs:attribute name="startOffset" type="xs:nonNegativeInteger"/>
        <xs:attribute name="endOffset" type="xs:nonNegativeInteger"/>
    </xs:complexType>

    <xs:complexType name="SourceElement" abstract="true">
        <xs:sequence>
            <xs:element name="Metadata" type="Metadata"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="Declarable" abstract="true">
        <xs:complexContent>
            <xs:extension base="SourceElement">
                <xs:sequence>
                    <xs:sequence id="modifiers" minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="Modifier" type="modifierType"/>
                    </xs:sequence>
                </xs:sequence>
                <xs:attribute name="id" type="xs:string"/>
                <xs:attribute name="returnType" type="xs:string"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Module">
        <xs:complexContent>
            <xs:extension base="Declarable">
                <xs:sequence>
                    <xs:sequence id="extends" minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="Extends" type="xs:string"/>
                    </xs:sequence>
                    <xs:sequence id="implements" minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="Implements" type="xs:string"/>
                    </xs:sequence>
                    <xs:sequence id="members" minOccurs="0" maxOccurs="unbounded">
```

```xml
                        <xs:element name="Member" type="Declarable"/>
                    </xs:sequence>
                </xs:sequence>
                <xs:attribute name="moduleType" type="moduleType"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Unit">
        <xs:complexContent>
            <xs:extension base="Declarable">
                <xs:sequence>
                    <xs:sequence id="parameters" minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="Parameter" type="Property"/>
                    </xs:sequence>
                    <xs:element minOccurs="0" name="Body" type="Expression"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Property">
        <xs:complexContent>
            <xs:extension base="Declarable">
                <xs:sequence>
                    <xs:element minOccurs="0" name="Initializer" type="Expression"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Expression">
        <xs:complexContent>
            <xs:extension base="SourceElement">
                <xs:sequence>
                    <xs:sequence id="innerScope" minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="Element" type="SourceElement"/>
                    </xs:sequence>
                </xs:sequence>

                <xs:attribute name="context" type="xs:string"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Loop">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:sequence id="evaluations" maxOccurs="unbounded">
                    <xs:element name="Evaluation" type="Expression"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Catch">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:sequence>
                    <xs:element name="Exception" type="Property"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>


    <xs:complexType name="Conditional">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:sequence>
                    <xs:element name="IfExpr" type="Expression"/>
                    <xs:sequence id="elseIfs" minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="ElseIfExpr" type="Expression"/>
                    </xs:sequence>
                    <xs:element minOccurs="0" name="ElseExpr" type="Expression"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Switch">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:sequence>
                    <xs:element name="Subject" type="Expression" minOccurs="0"/>
                    <xs:sequence id="switchCases" maxOccurs="unbounded">
                        <xs:element name="Case" type="SwitchCase"/>
                    </xs:sequence>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="SwitchCase">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:sequence>
                    <xs:element name="Pattern" type="Expression" minOccurs="0"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="LogicalSequence">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:sequence id="operands" minOccurs="2" maxOccurs="unbounded">
                    <xs:element name="Operand" type="Expression"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Jump">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:attribute name="label" type="xs:string"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Lambda">
```

```
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:sequence>
                    <xs:element name="Unit" type="Unit"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Access" abstract="true">
        <xs:complexContent>
            <xs:extension base="Expression">
                <xs:attribute name="declarableId" type="xs:string"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="ReferenceAccess">
        <xs:complexContent>
            <xs:extension base="Access">
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="UnitCall">
        <xs:complexContent>
            <xs:extension base="Access">
                <xs:sequence>
                    <xs:sequence minOccurs="0" maxOccurs="unbounded" id="unitCallArgs">
                        <xs:element name="Argument" type="Expression"/>
                    </xs:sequence>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Assignment">
        <xs:complexContent>
            <xs:extension base="Access">
                <xs:sequence>
                    <xs:element minOccurs="0" name="Value" type="Expression"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:simpleType name="moduleType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="class"/>
            <xs:enumeration value="enum"/>
            <xs:enumeration value="interface"/>
            <xs:enumeration value="record"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="modifierType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="public"/>
            <xs:enumeration value="protected"/>
            <xs:enumeration value="private"/>
            <xs:enumeration value="static"/>
```

```xml
            <xs:enumeration value="final"/>
            <xs:enumeration value="abstract"/>
            <xs:enumeration value="override"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Listing 55: LAMP Metamodel XSD