JOHANNES KEPLER
UNIVERSITY LINZ

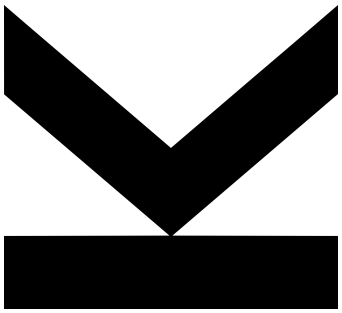
Author / Eingereicht von
Daniel Pfeffer
K12046745

Submission / Angefertigt am
Institute for System Software

Thesis Supervisor / First Supervisor / BeurteilerIn / ErstbeurteilerIn / ErstbetreuerIn
DI Dr. **Weninger Markus, BSc.**

Month Year / Monat Jahr

On the Applicability of Annotation-based Source Code Modification in Kotlin



Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Kurzfassung

Annotationen fügen Quellcode-Entitäten wie Klassen oder Funktionen Metadaten hinzu, die später von so genannten *Annotationsprozessoren* verarbeitet werden können, um z.B. den Quellcode des annotierten Programms zu verändern. Während Annotations Verarbeitung in Java ausreichend erforscht ist, fehlt der Kotlin-Community noch eine umfassende Zusammenfassung. Daher fassen wir in diesem Paper die wichtigsten in Kotlin verfügbaren Ansätze zusammen: (1) Verarbeitung von Annotationen zur Kompilierzeit mittels (a) Kotlin Annotation Processing Tool (KAPT), (b) Kotlin Symbolic Processing (KSP) oder (c) Eines benutzerdefinierten Kotlin Compiler Plugins; sowie (2) Änderung von Code zur Ladezeit mittels eines Agenten oder eines benutzerdefinierten "class loader". Wir stellen Proof-of-Concept-Implementierungen zur Verfügung, diskutieren Vor- und Nachteile und konzentrieren uns insbesondere darauf, wie gut jeder Ansatz die Modifikation des annotierten Quellcodes unterstützt. Dies soll Entwicklern und Forschern helfen, besser zu entscheiden, wann sie welchen Ansatz verwenden sollten.

Abstract

Annotations add metadata to source code entities such as classes or functions, which later can be processed by so-called *annotation processors* to, for example, modify the annotated program's source code. While annotation processing has been well-explored in Java, the Kotlin community still lacks a comprehensive summary. Thus, in this paper, we summarize the main approaches available in Kotlin: (1) Compile-time annotation processing using (a) Kotlin Annotation Processing Tool (KAPT), (b) Kotlin Symbolic Processing (KSP), or (c) writing a custom Kotlin Compiler plugin; as well as (2) load-time code modification using an agent or a custom class loader. We provide proof-of-concept implementations, discuss advantages and disadvantages, and specifically focus on how well each approach supports modifying the annotated source code. This should help developers and researchers to better decide when to use which approach.

Contents

1 Introduction

Various programming languages allow developers to provide additional metadata in the form of annotations. Frameworks such as JPA [1], Spring [2] or Kotless [3] heavily make use of this feature. There is a wide array of other application areas for annotations [4, 5], including testing [6, 7], GUI programming [8] or non-nullability annotating [9]

According to Yu et al. [10] *Java code with annotations tends to be less error-prone*, yet they also mention that *better tutorials and dedicated training would help novice developers [...] to more use annotations*. We want to tackle this lack of literature: While DBLP [11] currently lists 74 papers with the words *Java* and *Annotation* in their title, not a single one has a title containing *Kotlin* and *Annotation*. Even though this is just a rudimentary metric, a thorough web search confirmed our initial assumption: there is clearly a lack of guidance when it comes to developing annotation processors in Kotlin.

In this work-in-progress paper, we summarize Kotlin's existing approaches for annotation processing. We highlight their advantages and disadvantages and discuss how they can be used to *modify* the annotated source code. Based on the experience we gained while implementing a simple yet realistic annotation-based post-condition checking system, we provide a guideline on when to use which approach. Thus, our contributions are:

- an overview on how to process annotations in Kotlin, specifically at compile time using the Kotlin Annotation Processing Tool (KAPT) (section 4.1), Kotlin Symbolic Processing (KSP) (section 4.2), or a custom Kotlin compiler plugin (section 4.3); or at load time using an agent or a custom class loader (chapter 5).
  - This includes an example use case (chapter 3) implemented as a compiler plugin (compile-time modification) and an agent (load-time modification), see `https://github.com/Daniel-Pfeffer/Annotations-Source-Code-Modification`
- a thorough discussion of the advantages and disadvantages of each approach, as well as a guideline on when to use which approach (chapter 6)

2 Background

As all discussed approaches revolve around annotation processing and the programming language Kotlin, this section provides an overview of the relevant background.

2.1 Java Annotations and Their Processing

Annotations in Java have become an integral part of the language since they were introduced in Java 1.5 in 2004 as part of Java Specification Request (JSR) 175 [12]. They add metadata information primarily to indicate that an annotated element should be processed in a special way by development tools, deployment tools, or run-time libraries.

Annotations are either processed (1) at run time or (2) at compile time using the *Annotation Processing API* [13], an API that enables developers to plug custom annotation processors into `javac` (before Java 1.6, the *Annotation Processing Tool (apt)* [12, 14] was used for this task).

While annotation processors are typically tasked with writing new files or reporting error messages [15], also modifying the underlying source code is possible: (1) Load-time-based approaches (such as [16]) typically rely on a Java agent or a custom class loader to modify the annotated entities when they are loaded, while (2) annotation processors might go against their design philosophy and use the (quite complex) *Compiler Tree* API [17] to modify the compiled *Abstract Syntax Tree (AST)* as a workaround [18].

2.2 Kotlin

Kotlin is a modern, statically typed language developed by JetBrains. It was initially targeted to run on the Java Virtual Machine (JVM), enabling developers to write and run Kotlin code alongside existing Java code (or any other JVM-based language), and has since developed into a multiplatform language [19]. This allows Kotlin source to target Android, iOS, desktop, web, and native at the same time with only a minor target-specific source code.

3 Invariants Framework

In this section, we present an example that we used to test the various annotation processing approaches regarding their applicability for source code modification. In this example, a `Holds` annotation (see Listing 3.1) can be attached to properties or function parameters to define a condition that must hold at any time. Annotated entities should automatically be verified at all necessary locations, for example when a new value is assigned. Imagine we have an `Account` class that represents a bank account, we would be able to write code as shown in Listing 3.2, using verfiers as the one shown in Listing 3.3.

To achieve this, an annotation processor first has to find all annotated properties and parameters. For each annotated property, the processor then has to add verification code

- on initialization (which might lead to a new `init` block, as shown in Listing 3.2 (1)).
- after every value write (as shown in Listing 3.2 (3)). Writes can be found as calls to the property's setter function (which, together with a property's getter function, form the property's *property accessors* [20]).

For each annotated parameter, the relevant check has to be performed at the beginning of the function (as shown in Listing 3.2 (2))

```
@Target(PROPERTY, VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
annotation class Holds(val verifier: KClass<out Verification<*>>)
```

Listing 3.1: `Holds` annotation that can be attached to properties and function parameters.

```
class Account(@Holds(PositiveOrZero::class) var balance: Int) {
// (1) init { PositiveOrZero.verify(balance) }
fun withdraw(@Holds(Positive::class) amount: Int) {
// (2) Positive.verify(amount)
this.balance -= amount
// (3) PositiveOrZero.verify(balance)
```

```
} /* ... */
}
```

Listing 3.2: Using an annotation-based verification system, we do not need to call the verfication code (see comments) manually, as they are generated automatically by the annotation processor at the correct locations.

```
object PositiveOrZero : Verification<Int>{
override fun verify(toVerify: Int) {
if (toVerify < 0) { /* handle unsuccessful verification */ }
}
}
```

Listing 3.3: Verifier to ensure a given integer is positive or 0.

4 Compile-time Annotation Processing

In this section, we present three annotation processing techniques that run as part of the compilation process, namely the *Kotlin Annotation Processing Tool (KAPT)*, *Kotlin Symbolic Processing (KSP)*, as well as *custom Kotlin compiler plugins*.

4.1 Kotlin Annotation Processing Tool

The *Kotlin Annotation Processing Tool (KAPT)* was the first annotation processing tool developed by JetBrains [21]. It enables using existing Java annotation processors with Kotlin. This means that by using KAPT, an annotation processor written for Java can be used for Kotlin without modifications.

Example Due to the following limitations, we deliberately do not provide an example on how to develop further KAPT processors to not encourage bad practices. Also, Jetbrains suggests using KSP instead [21].

Limitation #1: Performance To use existing annotation processors defined for Java source code, KAPT has to generate Java stubs from Kotlin files. As stub generation is expensive, it has a significant impact on the compile time. According to an article by Uber regarding their performance of Kotlin Builds [22], Kotlin with KAPT adds an overhead of around 95 percent compared to compiling Kotlin without KAPT. In contrast, they state that Java with apt has an overhead of around 5 percent.

Limitation #2: Multi-Platform Kotlin Since KAPT is exclusive to the JVM platform, it cannot be used when targeting other platforms such as Kotlin's native backend.

Limitation #3: Kotlin Features Not Available Language features specific to Kotlin without a direct equivalent in Java are not directly accessible from within the annotation processors. This includes information about explicit nullability, primary constructors, properties, and so on. [23]

Limitation #4: Maintenance Mode Furthermore, since October 2021, KAPT has been in maintenance mode, meaning no new functionality will be introduced. How that will affect future performance improvements is not further discussed, but probably not to be expected.

Limitation #5: Complex Code Modification Since Java annotation processors (and thus also KAPT) are encouraged to not modify the code being compiled, only workarounds based on the complex Compiler Tree API exist [24].

4.2 Kotlin Symbolic Processing

The *Kotlin Symbolic Processing (KSP)* API — an abstraction of the more advanced *kotlinc* compiler plugin API that will be explained in section 4.3 – simplifies developing lightweight compiler plugins. It was developed to offer a powerful metaprogramming tool similar to KAPT, whilst being Kotlin idiomatic and integrated into the Kotlin compilation process. Thus, the existing problems with KAPT, especially the performance issues, are mitigated. For instance, JetBrains claims that KSP can run up to 2 times faster than KAPT. [25]

In contrast to one of KAPT's limitations — being tied to the JVM — KSP addresses all of Kotlin's targets, therefore being suitable for multiplatform projects.

Example A KSP processor has to implement the interface `SymbolProcessor` and provides, in its most basic form, a function `process(resolver: Resolver)` that will be called when the processor is run. The `resolver` can be used to obtain information about all source files, which in turn might be processed one-by-one, often using a `KSVisitor`. The visitor interface contains around 25 functions in the style of `visitXYZ(xyz: KSXYZDeclaration)` that can be overridden to perform certain operations when one of the respective code

elements is encountered. With a focus on annotation processing, the `resolver` also provides a function `getSymbolsWithAnnotation(...)` which can be used to easily obtain all elements annotated with a certain annotation. We can then, similar to reflection, query information about every annotated element (for example whether it is a property or a function parameter). A `CodeGenerator` instance can be used to create *new* code, typically through the help of a code generation framework such as *KotlinPoet* [26].

Limitation #1: Fine-grained Analysis KSP lacks the possibility to examine expression-level instructions. [27]

Limitation #2: Source-code Modification With KSP, it is not possible to modify source code. [27] This is further enforced by not even providing an AST-modification-based workaround, as available in Java annotation processors.

4.3 Compiler Plugins

As a final compile-time approach, we present custom Kotlin compiler plugins. First, we discuss how Kotlin's new K2 compiler is structured, using a frontend and backend design philosophy, each with its own intermediate representation. Following, we show that one can inject custom plugins at different compilation phases and present which kind of plugin we used to implement our example use case.

4.3.1 The K2 Kotlin Compiler

Kotlin's new K2 compiler aims primarily to accelerate the development of new language features by unifying certain tasks across all Kotlin Multiplatform targets. It also brings general performance improvements and should provide an official API for compiler plugins. As there was no need for different targets when Kotlin's first compiler was developed (since the only target was the JVM), when new targets were introduced (such as the JavaScript target), an entirely new backend had to be developed that builds the target code completely from scratch based on the syntax tree and its semantic information. With the new K2 compiler, that changed: A new frontend intermediate representation (FIR) has

been introduced, and all backends now rely on a common intermediate representation (IR). In a common backend, the K2 compiler performs general work (and optimizations) needed for every target based on this IR. Only entirely target-specific functionality has to be added to the target-specific backends.

Compiler Frontend The compiler frontend is split into four phases, each with different sub-phases, as shown in the top half of Figure 4.1. The parsing phase executes the *Lexical Analysis* to generate *tokens* which are then used for the *Syntax Analysis*, after which an *Abstract Syntax Tree (AST)* is created. Then, a *Semantic Analyzer* is invoked on the AST, after which the *Program Structure Interface (PSI)* alongside a *BindingContext*, i.e., a lookup table for semantic information, is created. In the intermediary code generation phase, this PSI is transformed to the FIR, which combines the traditional PSI and the BindingContext. The FIR is used in the resolution phase to perform *Code Analysis*, a last stage to report different diagnostics on the FIR and to modify the FIR according to frontend plugins. After the last code analysis stage reports success, the FIR is passed to the compiler backend. [28]

Compiler Backend The new common compiler backend (see bottom half of Figure 4.1) receives the FIR as input and converts it, using optional compiler plugins, to the *Intermediate Representation (IR)*. After the IR generation, the compiler backend invokes the common optimization phase. This optimized IR is then passed to a target-specific backend (for example the JVM backend), which transforms the optimized IR to the target-specific code (for example Java bytecode). Since most of the work now happens in the common compiler backend, the target-specific backends become smaller and therefore faster to adapt to new language features.

Plugins The K2 compiler offers native support to attach multiple plugins at the different phases in the compile process, ranging from frontend analysis to backend IR generation (suitable phases for compiler plugins are marked with a plus symbol (+) in Figure 4.1). Some of these plugin-able extensions only affect one target, for example, the `SyntheticJavaResolveExtension` [29] plugin is JVM-backend-specific and generates synthetic constructs only relevant there, but not in, for example, JavaScript. Nevertheless, most plugins operate on either the FIR or the IR, therefore being general. f A compiler plugin can, in turn, consist of multiple compiler plugins, and each plugin can be attached

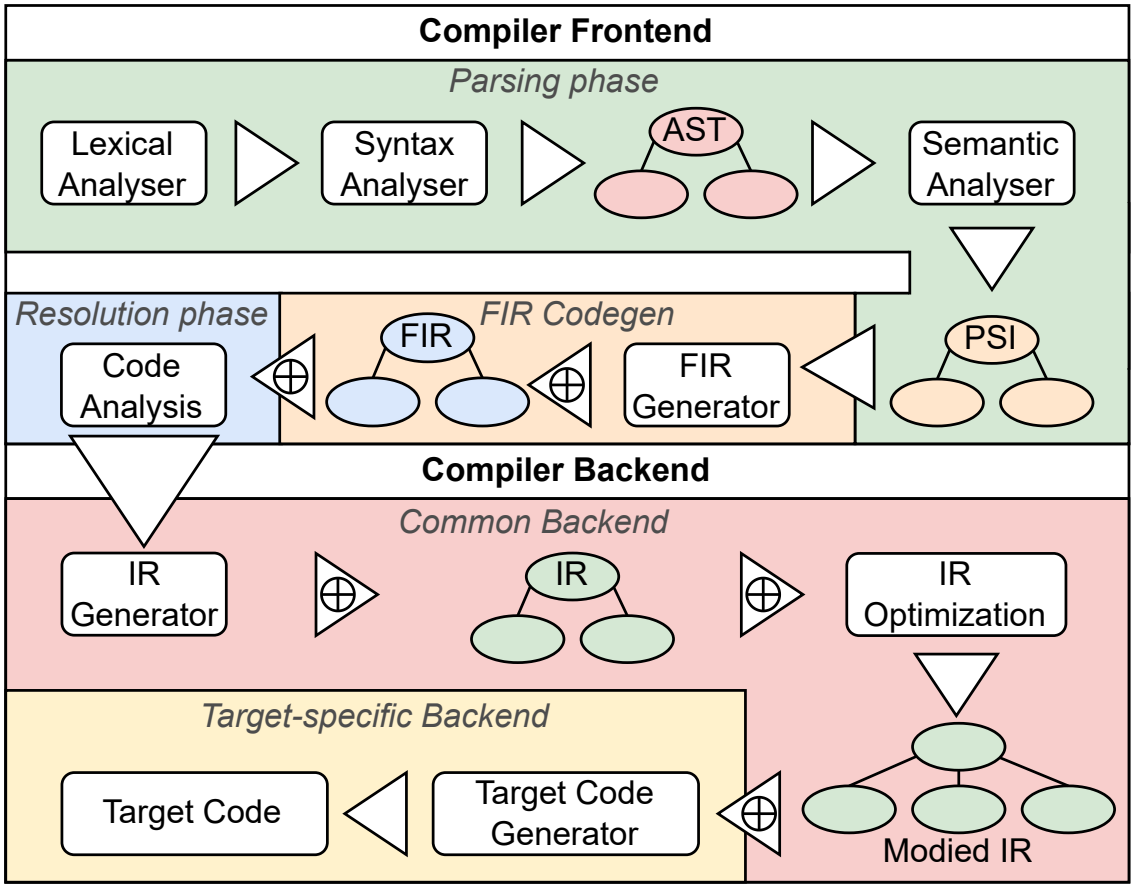


Figure 4.1: Simplified overview of Kotlin's K2 compiler compilation process.

to a different stage in the compilation process. For example, the prominent *Jetpack Compose* [30] compiler plugin registers multiple plugins to facilitate complex user interface creation. Two other well-known compiler plugins that change the default behavior of Kotlin are the *NoArg*-Plugin [31], which generates an additional zero-argument default constructor, and the *AllOpen*-Plugin [32], which automatically defines all classes as open (classes in Kotlin are `final` by default and have to be explicitly marked as `open` to be able to inherit from them).

4.3.2 Example

For the reference implementation of our example use case (cf. chapter 3), we decided to develop a common backend plugin that attaches to the *IR Generation* step (see Figure 4.1). This way, our plugin is not target-specific and thus can also be used with targets besides the JVM. To transform the IR, the processor has to implement the `IrGenerationExtension` interface. Similar to KSP, a *visitor* can be used to traverse all nodes in the IR tree from top to bottom. In contrast to KSP, when a compiler plugin visits an IR node, it is able to modify the underlying code.

The visitor visits the following functions for our use case: (A) `visitClassNew`, (B) `visitFunctionAccess`, (C) `visitConstructor`, (D) `visitFunctionNew` and (E) `visitAnonymousInitializerNew`. Each function is discussed in its respective paragraph.

`visitClassNew` This function is called when a class is visited for the first time, therefore, being new. Here we simply add an *anonymous initializer* if none should exist.

`visitFunctionAccess` This function is called when a function access expression, i.e., a function call, is encountered. If the called function is the setter of a property annotated with our `@Holds` annotation, we have to insert verification code, as is shown in Listing 4.1. At (1) we generate (and finally return) a new IR block, which in turn contains two function calls. The first function call (2) is the original call to the setter, i.e., the new value is written to the property. The `generateVerificationFunctionCall` function (3) generates the IR code for calling the respective verification function. For this, we pass the property, based on which the function checks if the element is annotated and verify-able. The second parameter of the `generateVerificationFunctionCall` function (4) generates a function call to the *getter* of the property to serve as an argument for the verification function call. In addition, we also have to give context, who calls the function (5), otherwise an incorrect function may be called, which in this case is the same receiver as the original expression. What basically happens in `generateVerificationFunctionCall` is that based on the annotation information `@Hold(XY::class)` it generates IR code that, simply speaking, executes `XY().verify(property)`. As an optimization, we cache the verifier objects such that only one of each is allocated. The unary plus symbol in front of (2)

and (3) is part of Kotlin's compiler plugin DSL and simply adds the two `IrExpressions` to the body of the enclosing `IrBlock`.

Listing 4.1: A simplified version of how we generate verification calls after a property write.

```
return DeclarationIrBuilder(context, expression.symbol).irBlock { // (1)
    +super.visitFunctionAccess(expression) // (2)
    +generateVerificationFunctionCall( // (3)
        property, /* (4) */irCall(property.getter!!).also {
            it.dispatchReceiver = expression.dispatchReceiver // (5)
        }
    )
}
```

`visitConstructor` This function is called when a constructor is encountered. As Kotlin allows for properties being declared in the primary constructor [33], and we allow annotations to affect function parameter, the annotation defaulting [34] assumes that the annotation is suited for the function parameter instead of the class property, therefore we have to correct the behavior wherever applicable, to simplify the behavior in subsequent IR code generation.

`visitFunctionNew` This function is called when a function is first declared in the IR. For each function, we iterate the list of parameters, and if a parameter is annotated with our `Holds` annotation, the verification code is added at the beginning of the function's body.

`visitAnonymousInitializerNew` This function is called when a new anonymous initializer, `init`, is encountered. For the `init` block, for every annotated property of the parent class, a verification function call is generated and added to the statements of the `init` call.

It is noteworthy to mention that as the visitor visits top to bottom in the IR tree, all functions are visited after the class and after the primary constructor invocation. Therefore, it is possible to correct, for example, the annotation use-site target defaulting of Kotlin without a problem.

But as the rest of the visitation order depends on the order of declaration in the source code, no IR transformations should depend on previous transformations that can be on the same IR level.

Once all IR nodes were visited, the compiler simply passes the modified IR to another *IR Generation* compiler plugin or, if no such plugin exists, to the *IR Optimization* phase.

A simplified example If we assume the following source code snippet

Listing 4.2: Original source code using our compiler plugin

```
data class BankAccountView(@Holds(IsPositive::class) val amount: Int, val holder: String) {
    fun canWithdraw(@Holds(IsPositive::class) withdrawAmount: Int): Boolean =
        amount >= withdrawAmount
}
```

the Kotlin compiler first generates an IR tree, simplified in Figure 4.2.

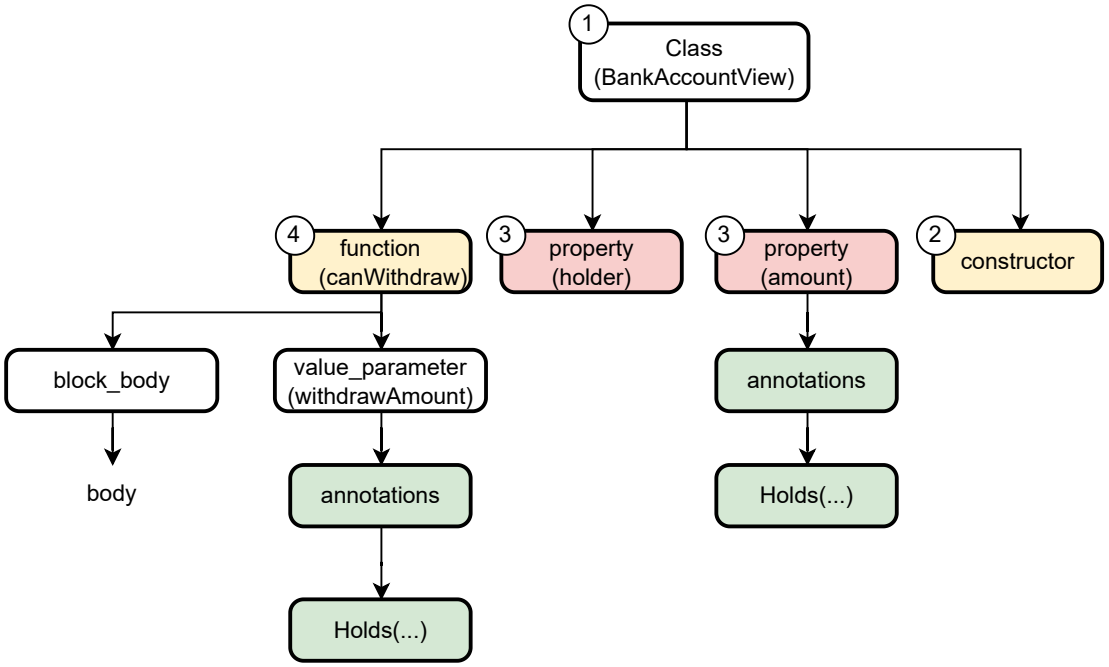


Figure 4.2: Simplified IR Tree before our compiler plugin—an example

Our compiler plugin then begins traversing the tree with `visitClassNew` (1), where the plugin creates a new ***empty*** *anonymous initializer* if none should exist.
Afterwards, the compiler plugin visits `visitConstructor` (2). For the simplied example, we assume the properties (3) already received the correct annotation, therefore nothing happens in `visitConstructor`. Next, the compiler plugin visits `visitFunctionNew` (4). Here the plugin adds the verification code to the function body, which is shown in Figure 4.3 (B) and appends the original function body after the verification statement. Lastly, the compiler plugin visits `visitAnonymousInitializerNew`, which was created in (1). There the compiler plugin generates the verification code for the *amount* property (3), as shown in Figure 4.3 (A).

After the compiler plugin has finished with the visitAnonymousInitializerNew function, the modified IR tree, as shown simplified in Figure 4.3 is passed along the compiler pipeline, therefore either to another compiler plugin or to the IR optimization phase.

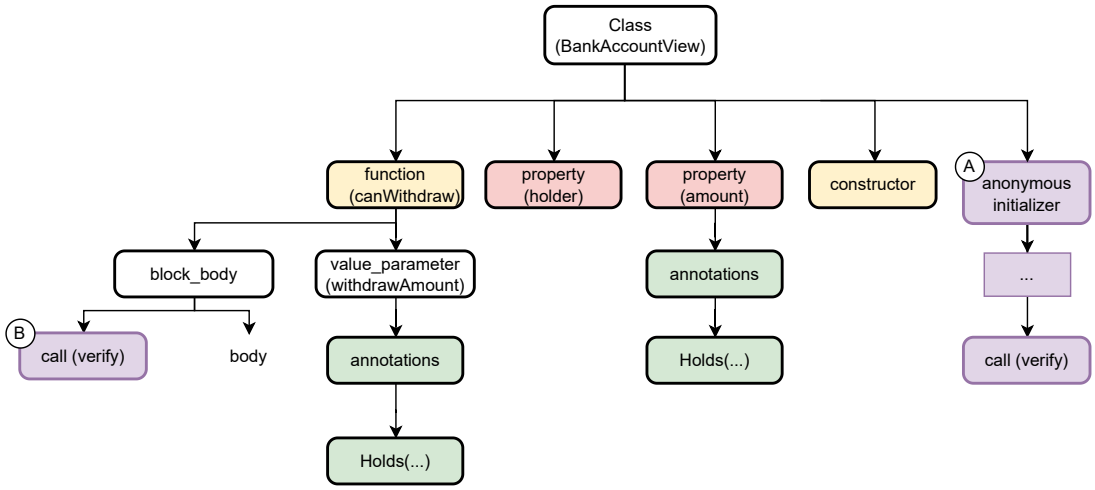


Figure 4.3: Simplified IR Tree after our compiler plugin—an example

4.3.3 Limitations

While compiler plugins are the most powerful and versatile approach for annotation-based source code modification, they are not without limitations.

Limitation #1: Lack of Documentation Currently, the main resources for custom compiler plugin development are videos by JetBrains employees (for example [35]), written tutorials by third parties (e.g., [36, 37]), and examples on GitHub (for example [38, 39]).

Limitation #2: Kotlin-specific To the best of our knowledge, Kotlin compiler plugins can only process Kotlin code, thus, for example, Java code cannot be processed with compiler plugins.

Limitation #3: Order of Plugin Execution Multiple compiler plugins might affect each other when executing in a different order, i.e., different code could be generated if plugin *A* runs before plugin *B* or after plugin *B*.

4.3.4 Usage

Using a Kotlin compiler plugin is, when defined correctly, relatively easy as long as a Gradle plugin is provided. Otherwise, the plugin has to be manually configured and attached to the Kotlin compiler. With the added Gradle plugin, the implemented plugin can be added to any Gradle or Maven dependency script, therefore integrating directly with the build system. Thus, the compiler plugin simply has to be added to the plugin section of a Gradle project file and the complete initialization and appending to the compiler is done by Gradle, as shown in Listing 4.3.

Listing 4.3: Example of a Kotlin compiler plugin dependency added to a gradle build script

```
plugins {
    kotlin("jvm") version "1.8.20"
    application
    id("social.xperience.invariants") version "1.0"
}

dependencies {
    implementation("social.xperience:invariants-annotation:1.0")
}
```

If no such plugin is provided, or the project is not using Gradle, the compiler plugin has to be added to the compiler manually. This can be done by directly appending the compiler plugin to the CLI call of the Kotlin compiler as can be seen in Listing 5.2.

Listing 4.4: Example of a Kotlin compiler plugin dependency added to the CLI

```
kotlinc test.kt -include-runtime -d -Xplugin=pathToPlugin.jar
test.jar
```

The main difference between applying compiler plugins via Gradle or manually is that it's easier to deal with complex compiler plugins due to the added configuration options. This is especially helpful when working with more complicated setups, such as multiplatform projects.

5 Load-time Annotation Processing

This section presents the two dominating ways of modifying code at load time: Java agents and custom class loaders.

5.1 Java Agents

Java agents leverage the *Java Instrumentation API* to modify classes at load time or run time. This modification is performed on the bytecode level, often utilizing Java bytecode manipulation libraries such as *Javassist* [40, 41] or *ASM* [42].

5.1.1 Example

Java agents use a so-called *transformer* to modify a class's bytecode. A transformer must implement the `ClassFileTransformer` interface, which defines a `byte[] transform(...)` method. This method receives, besides other parameters, the name and the bytecode (as `byte[]`) of the class currently being transformed, and it must return the modified bytecode.

Our transformer utilizes Javassist, where the class currently being transformed is represented as a `CtClass` [43] object. It provides methods such as `getDeclaredFields` and `getDeclaredMethods`, which we use to find annotated properties and annotated function parameters. Javassist enables developers to add new bytecode by just providing the new source code as a simple string (compared to, for example, complex IR nodes as is the case with compiler plugins). This means that for a given annotated property with its setter method `s`, one can simply write `s.insertAfter(str)`, where `str` might contain the code

string `"verifierObj. verify(this.propName);"`. Therefore, the creation of the verification logic is compared to the compiler plugin approach easier, but due to the lacking compile time information finding where to add the verification logic is significantly more time-consuming. For instance, when checking if a given Kotlin property has an annotation, there are multiple ways where the annotation can be found, depending on the annotation's use-site target. With our implementation, there are four use-site targets that the annotation can reside in. These are as follows, for a primary constructor argument, for a Java field directly, for the `getter` of the respective java field, or there can be a custom function that provides the annotation, which is the default case for Kotlin properties. Therefore, to check if a Kotlin property was annotated, the Java Agent has to validate all these use-site targets to be sure that there exists an annotation for a given Java field. This is further complicated by the mentioned lack of compile time information as there exists, for example, no clear link between Java fields and their respective `setter/getter` except for the name of the respective `setter/getter` which is shown in Listing 5.1.

Listing 5.1: How to find either the annotated getter or the Kotlin generated annotation getter function

```
private fun CtField.getPropertyAnnotation(annotation: KClass<*>): Annotation? {
    return declaringClass.declaredBehaviors.firstOrNull { behaviour -> // (1)
        behaviour.name.startsWith("get${name.capitalized}")/*(2)*/
            && behaviour.hasAnnotation(annotation.java) /*(3)*/
    }?.getAnnotation(annotation.java) as Annotation? // (4)
```

In (1) we query through every existing declared behavior, i.e., all existing constructors and methods including private behavior, which sub-sequentially is (2) checked if it is the `getter` for the field or for the annotation and (3) if it is annotated with a specific annotation. If such a behavior is found (4) we return the annotation.

Unlike the compiler plugin approach, the Java agent approach, modifies the already existing setter instead of injecting the verification procedure calls directly at the source of the modification. Additionally, unlike the compiler plugin, instead of having one global cache for all verification objects, each class must have its own cache of used verification objects. As at no point during bytecode modification, it is for certain that no additional changes may happen, therefore, a global verification cache would never be able to be loaded without major performance drawback by repeatedly recompiling the verification cache.

5.2 Custom Class Loader

It is possible to achieve a similar effect by writing a custom class loader. In the custom class loader's `findClass` function, the class's byte code might be modified similarly as it would be using a Java agent (e.g., using a library such as Javassist). Thus, it is possible to check a class for the existence of certain annotations within it and, if so, modify the required code parts while loading that class.

Using the Java property `java.system.class.loader` one can change the system class loader to a specific class loader. Providing a custom system class loader allows us to automatically handle the loading (and modifying) of every class that might be annotated with our annotation(s).

5.3 Limitations

Limitation #1: Startup Time : Since the modifications happen at load time, this approach can slow down the startup of the application. Depending on the use case, this might be negligible, for example for long-running processes, or a real concern, for example for Android applications that rely on fast startup for high user experience.

In a minimal test setup (5 classes, 14 annotations), we noticed a startup time of about 40ms compared to a startup time of 10ms using a Custom compiler plugin. Nevertheless, we want to mention we did not implement any test cases that focus specifically on performance, neither did we focus on implementing the most performing agent.

Limitation #2: JVM-specific Since the presented load-time approaches are JVM-specific, they cannot be used to modify code generated by other backends.

Limitation #3 (agent-specific): Order of agent execution Similar to the compiler plugin approach, multiple agents might change the bytecode in different ways depending on their order of execution.

Limitation #4 (class-loader-specific): Multiple class loaders A class is not loaded by multiple class loaders but only by a single one. It is thus not possible for multiple class loaders to modify the same class.

5.3.1 Usage

Using a Java agent or class loader is straightforward as the agent or the class loader is just a jar file passed to the JVM using the respective `-javaagent`/`-Djava.system.class.loader` command line argument, as shown in

Listing 5.2: Example of a Java agent or class loader added to the JVM using the CLI

```
java -javaagent=pathToAgent.jar test.jar
java -Djava.system.class.loader=pathToLoader.jar test.jar
```

6 Discussion

As we have shown, each approach has its set of limitations. As the approaches differ greatly in the way they are implemented, it is hardly possible to switch from one to another one during development without heavy refactoring. Therefore, Table 6.1 reiterates the *advantages and disadvantages* of each approach, we discuss certain *performance* considerations, and we present a set of *guidelines* to aid in the decision-making process on when to use which approach.

Table 6.1: Advantages and disadvantages of Kotlin's different annotation processing techniques.

	Advantages	**Disadvantages**
KAPT (*maintenance mode*)	+ Can reuse Java processors	- Slow - No multiplatform - Not Kotlin-idiomatic - Complicated code mod.
KSP	+ Kotlin-idiomatic + Fast + Multiplatform	- No expr.-level analysis - No code modification
Compiler Plugins	+ Same as KSP but more versatile (compilation phases, ...)	- Most complex approach - Lack of documentation
Load-time Approaches	+ Easy to implement + Good tool support (Javassist [40], ASM [42], ...)	- Slower startup - No multiplatform - Not Kotlin-idiomatic

Performance To preliminarily evaluate how much overhead our custom compiler plugin and Java agent introduce (which have not been specifically implemented with high performance in mind), we developed a dummy application that simulates server-side business logic. It contains 23 classes, including verification procedures, with an overall number of 6 methods that perform dummy business logic. Within these, we annotated 25 fields and parameters with our `Holds` annotation.

Table 6.2: KAPT and KSP: Taken from related work. Plugin: We performed 2000 compilations (1000x without plugin (*off*), 1000x with plugin (*on*)). On average, our plugin increased the compile time by about 6%. Agent: We ran the application 2000 times (1000x with manually added verification code, 1000x with agent). On average, our agent increased the run time by 385 ms.

	Run Time
KAPT (compile time)	Uber: 95% overhead compared to pure Kotlin only [22]
KSP (compile time)	JetBrains: up to 2 times faster than kapt [25]
Compiler Plugin (compile time in ms) ***95% conf. interval***	off: [542, 544] (avg. 543ms) on: [575, 578] (avg. 576ms, +6.08%)
Java Agent (run time in ms) ***95% conf. interval***	off: [268, 268] (avg. 268ms) on: [652, 653] (avg. 653ms, +385ms)

Table 6.2 shows our results, as well as some statements regarding KAPT and KSP from other sources (since we do not have implementations for these approaches).

Guideline We suggest suitable Kotlin annotation processing approaches based on the following guidelines, also depicted in Figure 6.1:

- **No need to modify source code:** *KSP* (high performance; well-documented, de-facto standard for annotation processing in Kotlin)
- **Need to modify source code:**
  - **Startup time relevant:** *Custom Kotlin compiler plugin* (versatile, powerful capabilities; yet complex and documentation nearly non-existent)
  - **Startup time not critical (JVM-backend only):** *Load-time-based approach* using a custom class loader or an agent (easier to implement; helpful libraries and examples available)

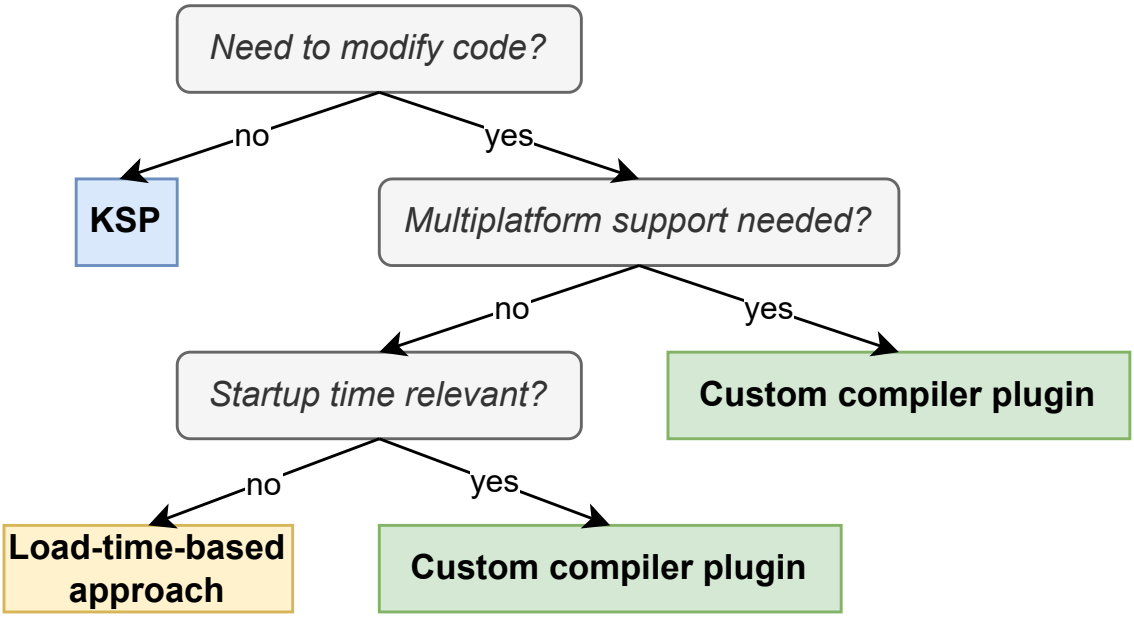


Figure 6.1: Decision tree for when to use which annotation processing technique in Kotlin.

6.1 Related Work

In the following, we present various related work that modifies source code based on annotations as well as resources for developing processors in KAPT, KSP as well as Kotlin compiler plugins.

Annotation-based Source Code Modification *Spoon* [44, 45] is a Java library to analyze, rewrite, transform, and transpile Java source code that utilizes compile-time annotation processing. Likewise, *Project Lombok* [18] leverages Java annotations to automatically generate boilerplate code such as `getter` and `setter` functions during compilation. The *Java Persistence API (JPA)* [46] as well as other `jakarta` packages use annotation processing for various tasks, including annotating ORM entities [46] to dependency injection [47]. Birillo et al. [48] present *Reflekt*, a tool to replace run-time reflection with compile-time reflection to improve performance. It relies on a Kotlin compiler plugin to modify the code being compiled.

KAPT, KSP and Kotlin Compiler Plugin Development Most introductory works that teach KAPT, KSP, and especially compiler plugins are distributed as videos [35], written tutorials [36, 37], or as plain code examples on GitHub [38, 39]. Most articles present a single, small example, without (many) references or resources to follow upon. In our experience, we achieved the best results by looking at existing projects such as *MapStruct* (a KAPT / Java Annotation Processer) [49], Google's *KSP examples* [50], or the *AllOpen* [51] and *NoArg* [52] compiler plugin.

Using Kotlin's Techniques to Extend Existing Work For Java, ample work presents approaches that either create new files or modify existing code. Such approaches could also be introduced in Kotlin: Those that create new files could be implemented with KSP, while approaches that want to hook deeper into the compilation process, for example to modify the bytecode produced by the compiler, could use a custom compiler plugin. In the following, we present a few selected examples.

Pattern-Based Structural Expressions (PBSE) [53, 54] are a way to automatically add annotations to various code entities of a yet unannotated Java codebase (besides other languages). Using PBSE's DSL, one can define a PBSE file that describes which code entities (for example which methods) should become annotated with which annotations. This description has similarities to a *pointcut* description in AspectJ's aspected-oriented programming approach [55, 56]. For example, one can define that every `public` method whose name starts with *test* should become annotated with JUnit's *@Test* annotation [57]. In simple terms, the tool takes a PBSE file (describing which annotations should be added where) alongside the (unannotated) Java source code to generate files that contain the annotated Java source code. PBSE files enable developers to use the same descriptions across multiple applications, and by exchanging the PBSE file one might switch to another framework without heavy refactoring. Since the existing code is not modified but only new code is generated (basically using string concatenation), a PBSE tool could be re-implemented in Kotlin using KSP. Since it would be enough to include the annotations in the bytecode during compilation without creating annotated Java source code at all, one could even develop a custom compiler plugin that completely hides the annotation process from the user.

Song and Tilevich [58] present a similar DSL to describe *metadata invariants*. These must hold for all matching source code entities, otherwise the developer is warned about the

non-conforming code locations. For example, one could define that all methods whose name starts with *test* must have the annotation `@Test` attached to them. They also present algorithms that, based on a given codebase with annotated entities, can derive possible invariant candidates. To analyze the codebase (either for inferring or checking), they walk the classes' abstract syntax trees using JDT [59]. Both, KSP or a custom Kotlin compiler plugin, could be used to perform the same tasks since no code has to be modified or generated and both techniques are able to inspect the syntax tree.

Tansey and Tilevich [60] present an approach that automatically upgrades source code according to a set of *when-then transformation rules* (which might be user-defined or automatically inferred from code examples). For example, they used their tool to upgrade a large codebase to a newer version of JUnit that used (different) annotations. Song et al. [61] extended this approach to also be able to upgrade legacy programs that were initially configured using XML but changed to an annotation-based configuration [60]. These transformations are purely text-based, thus KSP in combination with a code generation framework such as CodePoet could provide an excellent foundation to recreate these tools in Kotlin.

6.2 Limitations and Future Work

Even though being a work-in-progress paper, the paper strives to provide a comprehensive overview of annotation-based source code modification in Kotlin. To provide others with templates to develop their own code-modifying annotation processors, we host reference implementations for the, in our opinion, most flexible approaches: (1) a custom compiler-plugin and (2) an instrumenting agent. In the future, we might also provide reference implementations for the other presented approaches: KAPT (is in maintenance mode), KSP (no source code modification possible), and using a custom class loader (quite similar to an agent).

As we have discussed, the complexity of compiler plugin development and the lack of documentation makes it hard to even get started. Existing introductory compiler plugins [51, 52] are, generally speaking, often too simple, while more comprehensive compiler plugins[62, 63] are lacking explanations. One framework that claims to make plugin development easier, ARROW Meta [64], unfortunately has no working documentation

available. Thus, in the future, we want to extend and document our existing compiler plugin example in even more detail. With it, we want to provide a comprehensive resource for others wanting to learn about compiler plugin development and all of its phases.

To identify the needs of the Kotlin community in more detail, we also plan to perform a study on how annotations in Kotlin are currently used "in the wild", for example by mining repositories on GitHub [65, 66]. This might provide insights into which aspects of annotation-oriented development are most relevant to the community.

6.3 Conclusion

Due to a lack of existing work, we presented how annotations can be used to modify Kotlin source code. We explored three different compile-time approaches, namely using (1) Kotlin's Annotation Processing Tool (*KAPT*), (2) Kotlin's Symbolic Processing (*KSP*) or (3) custom Kotlin compiler plugins, as well as two different load-time approaches, namely using (1) an instrumenting Java agent or (2) a custom class loader. We highlighted the limitations of each of these approaches, especially that *KAPT* is already in maintenance mode and not actively developed anymore, as well as *KSP*'s constraint of only being able to *generate additional* source code files but not to being able to *modify* existing one.

We provide a reference annotation processor implementation for both, a custom Kotlin compiler plugin as well as a Java agent. These reference implementations, as well as our discussion on when to use which approach, will hopefully help developers and researchers to easier get started with annotation-based source code modification in Kotlin and might spark ideas on how to improve existing annotation processing tools and frameworks.

List of Figures

List of Tables

Bibliography

[1] Catalin Tudose and Carmen Odubasteanu. "Object-relational Mapping Using JPA, Hibernate and Spring Data JPA". In: *23rd International Conference on Control Systems and Computer Science, CSCS 2021*. IEEE, 2021, pp. 424–431. DOI: 10.1109/CSCS52396.2021.00076. URL: https://doi.org/10.1109/CSCS52396.2021.00076 (cit. on p. 1).

[2] Gabriel Menezes, Bruno B. P. Cafeo, and André C. Hora. "How are framework code samples maintained and used by developers? The case of Android and Spring Boot". In: *J. Syst. Softw.* 185 (2022), p. 111146. DOI: 10.1016/j.jss.2021.111146. URL: https://doi.org/10.1016/j.jss.2021.111146 (cit. on p. 1).

[3] Vladislav Tankov, Yaroslav Golubev, and Timofey Bryksin. "Kotless: A Serverless Framework for Kotlin". In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. IEEE, 2019, pp. 1110–1113. DOI: 10.1109/ASE.2019.00114. URL: https://doi.org/10.1109/ASE.2019.00114 (cit. on p. 1).

[4] Eduardo M. Guerra et al. "Idioms for code annotations in the Java language". In: *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs, SugarLoafPLoP 2010*. ACM, 2010, 7:1–7:14. DOI: 10.1145/2581507.2581514. URL: https://doi.org/10.1145/2581507.2581514 (cit. on p. 1).

[5] Eduardo Guerra. "Design Patterns for Annotation-Based APIs". In: *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*. SugarLoafPLoP '16. Buenos Aires, Argentina: The Hillside Group, 2016 (cit. on p. 1).

[6] Panagiotis Louridas. "JUnit: Unit Testing and Coding in Tandem". In: *IEEE Softw.* 22.4 (2005), pp. 12–15. DOI: 10.1109/MS.2005.100. URL: https://doi.org/10.1109/MS.2005.100 (cit. on p. 1).

[7] Viera K. Proulx and Weston Jossey. "Unit test support for Java via reflection and annotations". In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009*. ACM, 2009, pp. 49–56. DOI: 10.1145/1596655.1596663. URL: https://doi.org/10.1145/1596655.1596663 (cit. on p. 1).

[8] Mostafa Mehrabi, Nasser Giacaman, and Oliver Sinnen. "Unobtrusive Support for Asynchronous GUI Operations with Java Annotations". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018*. IEEE Computer Society, 2018, pp. 405–414. DOI: 10.1109/IPDPSW.2018.00075. URL: https://doi.org/10.1109/IPDPSW.2018.00075 (cit. on p. 1).

[9] Laurent Hubert. "A non-null annotation inferencer for Java bytecode". In: *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'08*. ACM, 2008, pp. 36–42. DOI: 10.1145/1512475.1512484. URL: https://doi.org/10.1145/1512475.1512484 (cit. on p. 1).

[10] Zhongxing Yu et al. "Characterizing the Usage, Evolution and Impact of Java Annotations in Practice". In: *IEEE Trans. Software Eng.* 47.5 (2021), pp. 969–986. DOI: 10.1109/TSE.2019.2910516. URL: https://doi.org/10.1109/TSE.2019.2910516 (cit. on p. 1).

[11] Michael Ley. "The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives". In: *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002*. Ed. by Alberto H. F. Laender and Arlindo L. Oliveira. Vol. 2476. Lecture Notes in Computer Science. Springer, 2002, pp. 1–10. DOI: 10.1007/3-540-45735-6_1. URL: https://doi.org/10.1007/3-540-45735-6%5C_1 (cit. on p. 1).

[12] Joshua Bloch. *JSR 175: A Metadata Facility for the JavaTM Programming Language*. 2002. URL: https://jcp.org/en/jsr/detail?id=175 (visited on 06/29/2023) (cit. on p. 2).

[13] Joe Darcy. *JSR 269: Pluggable Annotation Processing API*. 2005. URL: https://jcp.org/en/jsr/detail?id=269 (visited on 06/29/2023) (cit. on p. 2).

[14] Oracle Joe Darcy. *JEP 117: Remove the Annotation-Processing Tool (apt)*. 2011. URL: https://openjdk.org/jeps/117 (visited on 06/29/2023) (cit. on p. 2).

[15] Oracle. *JavaDoc: Interface ProcessingEnvironment*. 2023. URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/annotation/processing/ProcessingEnvironment.html (visited on 06/29/2023) (cit. on p. 2).

[16] Mark E. Royer and Sudarshan S. Chawathe. "Java unit annotations for units-of-measurement error prevention". In: *IEEE 8th Annual Computing and Communication Workshop and Conference, CCWC 2018*. IEEE, 2018, pp. 816–822. DOI: 10.1109/CCWC.2018.8301759. URL: https://doi.org/10.1109/CCWC.2018.8301759 (cit. on p. 2).

[17] Oracle. *Compiler Tree API*. 2023. URL: https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/index.html (visited on 06/29/2023) (cit. on p. 2).

[18] Project Lombok. *Project Lombok*. 2023. URL: https://projectlombok.org/ (visited on 06/29/2023) (cit. on pp. 2, 23).

[19] Svetlana Isakova. *Kotlin 1.4 Released with a Focus on Quality and Performance*. 2020. URL: %7Bhttps://blog.jetbrains.com/kotlin/2020/08/kotlin-1-4-released-with-a-focus-on-quality-and-performance%7D (visited on 06/29/2023) (cit. on p. 3).

[20] Jetbrains. *Kotlin: Properties - Getters and Setters*. 2023. URL: https://kotlinlang.org/docs/properties.html#getters-and-setters (visited on 06/29/2023) (cit. on p. 4).

[21] JetBrains. *Kotlin Annotation Processing Tool*. 2023. URL: https://kotlinlang.org/docs/kapt.html (visited on 06/13/2023) (cit. on p. 6).

[22] Uber. *Measuring Kotlin Build Performance at Uber*. 2019. URL: https://www.uber.com/blog/measuring-kotlin-build-performance/ (visited on 06/29/2023) (cit. on pp. 6, 22).

[23] Lukas Morawietz. *An Introduction to Kotlin Symbol Processing*. 2022. URL: https://www.codecentric.de/wissens-hub/blog/kotlin-symbol-processing-introduction (visited on 06/29/2023) (cit. on p. 7).

[24] Wang Zaijun. *Do You Know the Principle of Lombok That Has Been Used for a Long Time?* 2022. URL: https://www.alibabacloud.com/blog/599443 (visited on 06/29/2023) (cit. on p. 7).

[25] JetBrains. *Kotlin Symbol Processing API*. 2023. URL: https://kotlinlang.org/docs/ksp-overview.html (visited on 06/29/2023) (cit. on pp. 7, 22).

[26] Square Open Source. *KotlinPoet*. 2023. URL: https://square.github.io/kotlinpoet/ (visited on 06/29/2023) (cit. on p. 8).

[27] JetBrains. *KSP: Limitations*. 2023. URL: https://kotlinlang.org/docs/ksp-why-ksp.html#limitations (visited on 06/29/2023) (cit. on p. 8).

[28] Amanda Hinchman-Dominguez. *Kotlin Compiler Crash Course*. 2023. URL: https://github.com/Daniel-Pfeffer/Kotlin-Compiler-Crash-Course (visited on 06/29/2023) (cit. on p. 9).

[29] JetBrains. *SyntheticJavaResolveExtension source code*. 2021. URL: https://github.com/JetBrains/kotlin/blob/master/compiler/frontend.java/src/org/jetbrains/kotlin/resolve/jvm/extensions/SyntheticJavaResolveExtension.kt (visited on 06/28/2023) (cit. on p. 9).

[30] Google. *Jetpack Compose*. 2023. URL: https://developer.android.com/jetpack/compose (visited on 06/29/2023) (cit. on p. 10).

[31] JetBrains. *No-arg compiler plugin*. 2023. URL: https://kotlinlang.org/docs/no-arg-plugin.html (visited on 06/29/2023) (cit. on p. 10).

[32] JetBrains. *All-open compiler plugin*. 2023. URL: https://kotlinlang.org/docs/all-open-plugin.html (visited on 06/29/2023) (cit. on p. 10).

[33] JetBrains. *Kotlin Constructor Properties*. 2023. URL: htthttps://kotlinlang.org/docs/classes.html#constructors (visited on 11/10/2023) (cit. on p. 12).

[34] JetBrains. *Annotations use-site targets*. 2023. URL: htthttps://kotlinlang.org/docs/annotations.html#annotation-use-site-targets (visited on 11/10/2023) (cit. on p. 12).

[35] Mikhail Glukhikh. *K2 Compiler Plugins*. May 2023. URL: https://www.youtube.com/watch?v=Pl-89n9wDqo (cit. on pp. 15, 24).

[36] Ji Sungbin. *Say Hello to the Kotlin Compiler Plugin*. 2023. URL: https://betterprogramming.pub/say-hello-to-kotlin-compiler-plugin-f4e857be9a1 (visited on 06/29/2023) (cit. on pp. 15, 24).

[37] Brian Norman. *Writing Your Second Kotlin Compiler Plugin*. 2020. URL: https://bnorm.medium.com/writing-your-second-kotlin-compiler-plugin-part-1-project-setup-7b05c7d93f6c (visited on 06/29/2023) (cit. on pp. 15, 24).

[38] Brian Norman. *kotlin-power-assert compiler plugin*. 2023. URL: https://github.com/bnorm/kotlin-power-assert (visited on 06/29/2023) (cit. on pp. 15, 24).

[39] ktargeter. *ktargeter compiler plugin*. 2023. URL: https://github.com/ktargeter/ktargeter (visited on 06/28/2023) (cit. on pp. 15, 24).

[40] Shigeru Chiba. "Load-Time Structural Reflection in Java". In: *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*. 2000, pp. 313–336. DOI: 10.1007/3-540-45102-1_16. URL: https://doi.org/10.1007/3-540-45102-1%5C_16 (cit. on pp. 17, 21).

[41] Shigeru Chiba and Muga Nishizawa. "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators". In: *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*. 2003, pp. 364–376. DOI: 10.1007/978-3-540-39815-8_22. URL: https://doi.org/10.1007/978-3-540-39815-8%5C_22 (cit. on p. 17).

[42] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. "ASM: A code manipulation tool to implement adaptable systems". In: *Adaptable and Extensible Component Systems* 30.19 (2002) (cit. on pp. 17, 21).

[43] Shigeru Chiba. *Javassist: CtClass*. 2023. URL: https://www.javassist.org/html/javassist/CtClass.html (visited on 06/29/2023) (cit. on p. 17).

[44] Renaud Pawlak. "Spoon: Compile-time Annotation Processing for Middleware". In: *IEEE Distributed Syst. Online* 7.11 (2006). DOI: 10.1109/MDSO.2006.67. URL: https://doi.org/10.1109/MDSO.2006.67 (cit. on p. 23).

[45] Renaud Pawlak et al. "SPOON: A library for implementing analyses and transformations of Java source code". In: *Softw. Pract. Exp.* 46.9 (2016), pp. 1155–1179. DOI: 10.1002/spe.2346. URL: https://doi.org/10.1002/spe.2346 (cit. on p. 23).

[46] Eclipse Foundation. *Jakarta Persistence API*. 2022. URL: https://jakarta.ee/specifications/persistence/3.1/apidocs/ (visited on 06/29/2023) (cit. on p. 23).

[47] Eclipse Foundation. *Jakarta Context Dependency Injection API*. 2020. URL: https://jakarta.ee/specifications/cdi/3.0/apidocs/ (visited on 06/29/2023) (cit. on p. 23).

[48] Anastasiia Birillo et al. "Reflekt: a Library for Compile-Time Reflection in Kotlin". In: *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022*. IEEE, 2022, pp. 231–240. DOI: 10.1109/ICSE-SEIP55303.2022.9793932. URL: https://doi.org/10.1109/ICSE-SEIP55303.2022.9793932 (cit. on p. 23).

[49] MapStruct. *MapStruct annotation processor source code*. 2023. URL: https://github.com/mapstruct/mapstruct/tree/main (visited on 06/29/2023) (cit. on p. 24).

[50] Google. *KSP example source code*. 2023. URL: https://github.com/google/ksp/tree/main/examples/multiplatform (visited on 06/29/2023) (cit. on p. 24).

[51] JetBrains. *AllOpen compiler plugin source code*. 2023. URL: https://github.com/JetBrains/kotlin/tree/master/plugins/allopen (visited on 06/29/2023) (cit. on pp. 24, 25).

[52] JetBrains. *NoArg compiler plugin source code*. 2023. URL: https://github.com/JetBrains/kotlin/tree/master/plugins/noarg (visited on 06/29/2023) (cit. on pp. 24, 25).

[53] Eli Tilevich and Myoungkyu Song. "Reusable enterprise metadata with pattern-based structural expressions". In: *9th International Conference on Aspect-Oriented Software Development, AOSD 2010*. ACM, 2010, pp. 25–36. DOI: 10.1145/1739230.1739234. URL: https://doi.org/10.1145/1739230.1739234 (cit. on p. 24).

[54] Myoungkyu Song and Eli Tilevich. "Reusing metadata across components, applications, and languages". In: *Sci. Comput. Program.* 98 (2015), pp. 617–644. DOI: 10.1016/j.scico.2014.09.002. URL: https://doi.org/10.1016/j.scico.2014.09.002 (cit. on p. 24).

[55] Gregor Kiczales et al. "An Overview of AspectJ". In: *15th European Conference on Object-Oriented Programming, ECOOP 2011*. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 327–353. DOI: 10.1007/3-540-45337-7_18. URL: https://doi.org/10.1007/3-540-45337-7%5C_18 (cit. on p. 24).

[56] Gregor Kiczales et al. "Getting started with ASPECTJ". In: *Commun. ACM* 44.10 (2001), pp. 59–65. DOI: 10.1145/383845.383858. URL: https://doi.org/10.1145/383845.383858 (cit. on p. 24).

[57] JUnit Team. *JUnit 5*. 2023. URL: https://junit.org/junit5/ (visited on 09/01/2023) (cit. on p. 24).

[58] Myoungkyu Song and Eli Tilevich. "Metadata invariants: Checking and inferring metadata coding conventions". In: *34th International Conference on Software Engineering, ICSE 2012*. IEEE Computer Society, 2012, pp. 694–704. DOI: 10.1109/ICSE.2012.6227148. URL: https://doi.org/10.1109/ICSE.2012.6227148 (cit. on p. 24).

[59] Eclipse Foundation. *Eclipse JDT (Java development tools)*. 2023. URL: https://projects.eclipse.org/projects/eclipse.jdt (visited on 09/01/2023) (cit. on p. 25).

[60] Wesley Tansey and Eli Tilevich. "Annotation refactoring: inferring upgrade transformations for legacy applications". In: *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008*. ACM, 2008, pp. 295–312. DOI: 10.1145/1449764.1449788. URL: https://doi.org/10.1145/1449764.1449788 (cit. on p. 25).

[61] Myoungkyu Song, Eli Tilevich, and Wesley Tansey. "Trailblazer: a tool for automated annotation refactoring". In: *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*. ACM, 2009, pp. 813–814. DOI: 10.1145/1639950.1640028. URL: https://doi.org/10.1145/1639950.1640028 (cit. on p. 25).

[62] JetBrains. *Kotlinx serialization source code*. 2023. URL: https://github.com/JetBrains/kotlin/tree/master/plugins/kotlinx-serialization (visited on 06/29/2023) (cit. on p. 25).

[63] AndroidX. *Jetpack Compose source code*. 2023. URL: https://github.com/androidx/androidx/tree/androidx-main/compose (visited on 06/29/2023) (cit. on p. 25).

[64] ARROW. *Arrow Meta*. 2023. URL: https://github.com/arrow-kt/arrow-meta (visited on 06/29/2023) (cit. on p. 25).

[65] Mohammad Al-Marzouq, Abdullatif Al-Zaidan, and Jehad Al-Dallal. "Mining GitHub for research and education: challenges and opportunities". In: *Int. J. Web Inf. Syst.* 16.4 (2020), pp. 451–473. DOI: 10.1108/IJWIS-03-2020-0016. URL: https://doi.org/10.1108/IJWIS-03-2020-0016 (cit. on p. 26).

[66] Egor Spirin et al. "PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code". In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 13–17. DOI: 10.1109/MSR52588.2021.00014. URL: https://doi.org/10.1109/MSR52588.2021.00014 (cit. on p. 26).