



On the Applicability of Annotation-Based Source Code Modification in Kotlin (Work in Progress)

Daniel Pfeffer*

Institute for System Software
Johannes Kepler University Linz
Linz, Austria

Markus Weninger*

markus.weninger@jku.at
Institute for System Software
Johannes Kepler University Linz
Linz, Austria

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.

CCS Concepts: • **Applied computing** → **Annotation**; • **Software and its engineering** → **Source code generation**; Compilers; **Preprocessors**; Language features; • **General and reference** → **Surveys and overviews**; *Design*.

Keywords: Kotlin, Annotations, Source Code Generation, Source Code Modification, AST, Compiler Plugin

ACM Reference Format:

Daniel Pfeffer and Markus Weninger. 2023. On the Applicability of Annotation-Based Source Code Modification in Kotlin (Work in Progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Run-times (MPLR '23)*, October 22, 2023, Cascais, Portugal. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3617651.3622983>

*Both authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution 4.0 International License.

MPLR '23, October 22, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0380-5/23/10.

<https://doi.org/10.1145/3617651.3622983>

1 Introduction

Various programming languages allow developers to provide additional metadata in the form of annotations. Frameworks such as JPA [61], Spring [41] or Kotless [58] heavily make use of this feature. There is a wide array of other application areas for annotations [17, 18], including testing [38, 50], GUI programming [40] or non-nullability annotating [20].

According to Yu et al. [63] *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 [37] 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 (Section 5).
 - This includes an example use case (Section 3) implemented as a compiler plugin and an agent, see <https://bit.ly/annotation-modification-mplr-2023>
- a thorough discussion of the advantages and disadvantages of each approach, as well as a guideline on when to use which approach (Section 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 [5]. 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* [10], an API that enables developers to plug custom annotation processors into javac (before Java 1.6, the *Annotation Processing Tool (apt)* [5, 32] was used for this task).

While annotation processors are typically tasked with writing new files or reporting error messages [46], also modifying the underlying source code is possible: (1) Load-time-based approaches (such as [51]) 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* [45] to modify the compiled *Abstract Syntax Tree (AST)* as a workaround [49].

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 [21]. 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 Example Use Case

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 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 2, using verifiers as the one shown in Listing 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 2 (1)).
- after every value write (as shown in Listing 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* [26]).

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

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

Listing 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 2. Using an annotation-based verification system, we do not need to call the verification 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. 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. [25] 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 annotation processors using KAPT to not encourage bad practices.

Limitation #1: Maintenance Mode. Since October 2021, KAPT has been in maintenance mode. This means that there are no plans to implement new features for it. JetBrains suggests to use KSP instead. [25].

Limitation #2: Performance. To use existing Java annotation processors, 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 Uber [62], their Kotlin builds with KAPT experience an overhead of around 95 percent compared to pure Kotlin.

Limitation #3: Multiplatform. Since KAPT is exclusive to the JVM platform, it cannot be used when targeting other platforms such as Kotlin’s native backend.

Limitation #4: 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. [42]

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 [64].

4.2 Kotlin Symbolic Processing

The *Kotlin Symbolic Processing (KSP)* API – an abstraction of the more advanced *kotlinc* compiler plugin API that will be discussed in Section 4.3 – simplifies developing lightweight compiler plugins. It offers a powerful metaprogramming tool similar to KAPT, whilst being Kotlin idiomatic. 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. [27]

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 `KSVisor`. 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* [56].

Limitation #1: Fine-Grained Analysis. KSP lacks the possibility to examine expression-level instructions. [29]

Limitation #2: Source Code Modification. With KSP, it is not possible to modify source code. [29] This is further enforced by not even providing an AST-modification-based workaround, as available in Java annotation processors.

4.3 Compiler Plugins

As last 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 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. [19]

Compiler Backend. The new common compiler backend (see bottom half of Figure 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.

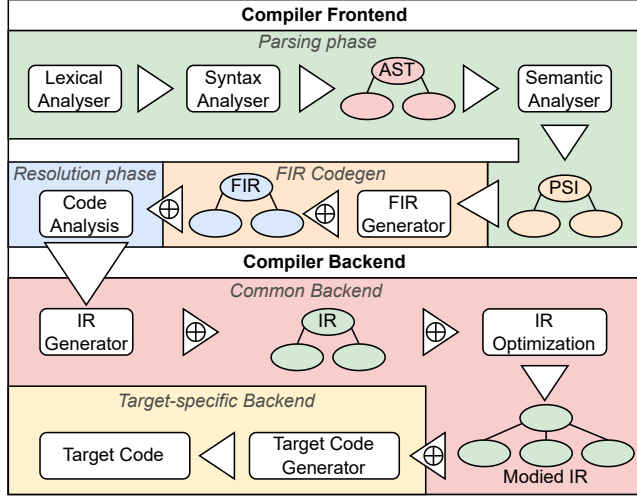


Figure 1. Simplified overview of Kotlin's K2 compiler compilation process.

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 1). Some of these plugin-able extensions only affect one target, for example, the *SyntheticJavaResolveExtension* [22] 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.

A compiler plugin can, in turn, consist of multiple compiler plugins, and each plugin can be attached to a different stage in the compilation process. For example, the prominent *Jetpack Compose* [15] 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* [30], which generates an additional zero-argument default constructor, and the *Allopen-Plugin* [23], 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 implementation of our example use case (cf. Section 3), we decided to develop a backend plugin that attaches to the *IR Generation* step (see Figure 1). This way, our plugin is not target-specific and can also be used with targets besides the JVM. Section A.1 contains the most important code constructs of our reference implementation.

To transform the IR in a backend plugin, we have to implement the *IrGenerationExtension* interface. Similar to KSP, a *visitor* can be used to traverse all IR nodes 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 first visitor function relevant to our use case is `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 outlined in Listing 4 (a simplified version of Listing 5). 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 `generateVerification` function (3) is responsible for generating the IR nodes to call the respective verification function. For this, we pass the annotated property as a parameter into `generateVerification`. Based on the property's annotation `@Holds(XY::class)`, the function generates and returns IR code that, simply speaking, executes `XY().verify(property)`. 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*.

```
return DeclarationIrBuilder(context, expression.symbol).irBlock { // (1)
    +super.visitFunctionAccess(expression) // (2)
    +generateVerification(property) // (3)
}
```

Listing 4. A simplified version of how we generate verification calls after a property write.

The second relevant visitor function is `visitFunctionNew`, which is called when a function is visited in the IR. We then iterate its parameters, and if a parameter is annotated with our `Holds` annotation, verification code is added at the beginning of the function's body (see Listing 6).

Lastly, we add an `init` block containing the respective verification code for annotated properties (see Listing 7). `init` blocks are automatically called during object instantiation.

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.

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 compiler plugin development are videos by JetBrains employees (e.g., [14]), written tutorials by third parties (e.g., [43, 57]), and examples on GitHub (e.g., [36, 44]).

Limitation #2: Kotlin-Specific. To the best of our knowledge, Kotlin compiler plugins can only process Kotlin code and not, for example, Java code.

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* [7, 9] or *ASM* [6].

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

Our transformer, whose most important code constructs can be found in [Section A.2](#), utilizes *Javassist*. In *Javassist*, the class currently being transformed is represented as a `CtClass` [8] object. A `CtClass` object provides methods such as `getDeclaredFields` and `getDeclaredMethods`, which we use to find annotated properties and annotated function parameters (see [Listing 8](#)). *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 for 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);"` (see in a similar fashion in [Listing 9](#)).

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: Multiple Class Loaders. A class can not be 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.

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 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 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 (<i>Javassist</i> [7], <i>ASM</i> [6], ...) | - 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 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).

Table 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 [62] |
| KSP (compile time) | JetBrains: up to 2 times faster than kapt [27] |
| 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) |

Guideline. We suggest suitable Kotlin annotation processing approaches based on the following guidelines, also depicted in Figure 2:

- **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)

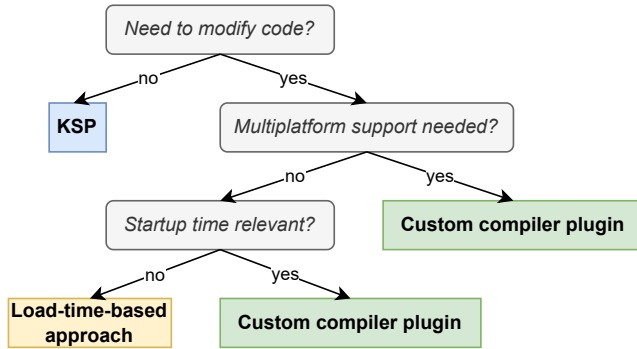


Figure 2. Decision tree for when to use which annotation processing technique in Kotlin.

7 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* [47, 48] is a Java library to analyze, rewrite, transform, and transpile Java source code that utilizes compile-time annotation processing. Likewise, *Project Lombok* [49] leverages Java annotations to automatically generate boilerplate code such as getter and setter functions during compilation. The *Java Persistence API (JPA)* [12] as well as other Jakarta packages use annotation processing for various tasks, including annotating ORM entities [12] to dependency injection [11]. Birillo et al. [4] 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 [14], written tutorials [43, 57], or as plain code examples on GitHub [36, 44]. 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 Processor) [39], Google’s *KSP examples* [16], or the *AllOpen* [24] and *NoArg* [31] 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, 60] 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 aspect-oriented programming approach [34, 35]. For example, one can define that every public method whose name starts with *test* should become annotated with JUnit’s *@Test* annotation [33]. 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 [52] 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 [13]. 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 [59] 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. [54] extended this approach to also be able to upgrade legacy programs that were initially configured using XML but changed to an annotation-based configuration [59]. 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.

8 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 [24, 31] are, generally speaking, often too simple, while more comprehensive compiler plugins [2, 28] are lacking explanations. One framework that claims to make plugin development easier, ARROW Meta [3], 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 [1, 55]. This might provide insights into which aspects of annotation-oriented development are most relevant to the community.

9 Conclusions

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.

A Sources

This appendix contains the most important code constructs used for our reference implementations of the example use case presented in Section 3. The full source code can be found at <https://bit.ly/annotation-modification-mplr-2023>.

A.1 Custom Kotlin Compiler Plugin

This section contains listings for our custom Kotlin compiler plugin. Listing 5 presents how to generate verification code after property sets, Listing 6 presents how to generate verification code for function parameters, and Listing 7 shows how we create an optional `init` (anonymous initializer) block.

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

Listing 5. The `DeclarationIrBuilder` is used to create a container containing (1) the originally intercepted expression, as well as (2) the generated verification call.

```

val body = func.body ?: return super.visitFunctionNew(func)
val valueParameterVerifications =
    DeclarationIrBuilder(context, func.symbol).irBlock { // (1)
        func.valueParameters.forEach {
            if (it.hasAnnotation(annotation)) {
                +generateVerificationFunctionCall(it, irGet(it))
            }
        }
    }
(body.statements as MutableList).add(0, valueParameterVerifications) // (2)

```

Listing 6. (1) The `DeclarationIrBuilder` is used to generate a block containing the verification call for all annotated value parameter. (2) This block is then added as the first element in the function body's statement list.

```

if (!declaration.hasAnonymousInitializer()) {
    declaration.declarations.add(
        irFactory.createAnonymousInitializer(
            /*...*/
        ).also {
            it.parent = declaration
            // empty body
            it.body = DeclarationIrBuilder(context, it.symbol).irBlockBody {
            }
        })
}

```

Listing 7. If the intercepted class does not have an `init` block, we add a new `init` block with an empty body which will be filled with the verification call at a later point.

A.2 Java Agent

This section contains listings for our custom Java agent. **Listing 8** presents how to find annotated Kotlin property getters in the bytecode, and **Listing 9** presents how to generate verification code for function parameters.

```

declaringClass.declaredBehaviors.firstOrNull { behavior ->
    behavior.name.startsWith(
        "get${name.capitalized}")
    ) && behavior.hasAnnotation(annotation.java)
}

```

Listing 8. When annotating a Kotlin property, the annotation can be attached to the backing field, the setter function or the getter function. To find the latter for a property named `name`, we iterate over all declared behaviors (methods + constructors) in the class, check whether the method name matches the getter name and whether the method is annotated.

```

// The entries contain all annotations (Array<Annotation>)
// of their respective function parameter
val params : Array<Array<Annotation>> = /* ... */

params.forEachIndexed { index, paramAnnotations ->
    paramAnnotations.forEach {

```

```

if (it.typeName == Commons.annotation.qualifiedName!!) {
    val parameter = ctMethod.getParameterNames()[index]
    val annotation: Holds = it.toAnnotationType(
        classPool.classLoader, classPool
    ) as Holds
    visitAnnotation(annotation)
    ctMethod.insertBefore("${ctClass.name}
        .${annotation.verifier.simpleName!!.lowercase()}
        .verify($parameter);")
}
}
}

```

Listing 9. By iterating over all annotations of all parameters of a function we can decide if a parameter has an applicable annotation. If so, we can then insert the verification code at the top of the function body.

References

- [1] Mohammad Al-Marzouq, Abdullatif Al-Zaidan, and Jehad Al-Dallal. 2020. Mining GitHub for research and education: challenges and opportunities. *Int. J. Web Inf. Syst.* 16, 4 (2020), 451–473. <https://doi.org/10.1108/IJWIS-03-2020-0016>
- [2] AndroidX. 2023. *Jetpack Compose source code*. <https://github.com/androidx/androidx/tree/androidx-main/compose>
- [3] ARROW. 2023. *Arrow Meta*. <https://github.com/arrow-kt/arrow-meta>
- [4] Anastasiia Birillo, Elena Lyulina, Maria Malysheva, Vladislav Tankov, and Timofey Bryksin. 2022. 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, 231–240. <https://doi.org/10.1109/ICSE-SEIP55303.2022.9793932>
- [5] Joshua Bloch. 2002. *JSR 175: A Metadata Facility for the Java™ Programming Language*. <https://jcp.org/en/jsr/detail?id=175>
- [6] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems* 30, 19 (2002).
- [7] Shigeru Chiba. 2000. Load-Time Structural Reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*. 313–336. https://doi.org/10.1007/3-540-45102-1_16
- [8] Shigeru Chiba. 2023. *Javassist: CtClass*. <https://www.javassist.org/html/javassist/CtClass.html>
- [9] Shigeru Chiba and Muga Nishizawa. 2003. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*. 364–376. https://doi.org/10.1007/978-3-540-39815-8_22
- [10] Joe Darcy. 2005. *JSR 269: Pluggable Annotation Processing API*. <https://jcp.org/en/jsr/detail?id=269>
- [11] Eclipse Foundation. 2020. *Jakarta Context Dependency Injection API*. <https://jakarta.ee/specifications/cdi/3.0/apidocs/>
- [12] Eclipse Foundation. 2022. *Jakarta Persistence API*. <https://jakarta.ee/specifications/persistence/3.1/apidocs/>
- [13] Eclipse Foundation. 2023. *Eclipse JDT (Java development tools)*. <https://projects.eclipse.org/projects/eclipse.jdt>
- [14] Mikhail Glukhikh. 2023. K2 Compiler Plugins. <https://www.youtube.com/watch?v=Pl-89n9wDqo>
- [15] Google. 2023. *Jetpack Compose*. <https://developer.android.com/jetpack/compose>
- [16] Google. 2023. *KSP example source code*. <https://github.com/google/ksp/tree/main/examples/multiplatform>
- [17] Eduardo Guerra. 2016. Design Patterns for Annotation-Based APIs. In *Proceedings of the 11th Latin-American Conference on Pattern Languages*

- of Programming (Buenos Aires, Argentina) (SugarLoafPLoP '16). The Hillside Group, USA, Article 9, 14 pages.
- [18] Eduardo M. Guerra, Menanes Cardoso, Jefferson O. Silva, and Clovis T. Fernandes. 2010. Idioms for code annotations in the Java language. In *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs, SugarLoafPLoP 2010*. ACM, 7:1–7:14. <https://doi.org/10.1145/2581507.2581514>
 - [19] Amanda Hinchman-Dominguez. 2023. *Kotlin Compiler Crash Course*. <https://github.com/Daniel-Pfeffer/Kotlin-Compiler-Crash-Course>
 - [20] Laurent Hubert. 2008. 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, 36–42. <https://doi.org/10.1145/1512475.1512484>
 - [21] Svetlana Isakova. 2020. *Kotlin 1.4 Released with a Focus on Quality and Performance*. <https://blog.jetbrains.com/kotlin/2020/08/kotlin-1-4-released-with-a-focus-on-quality-and-performance>
 - [22] JetBrains. 2021. *SyntheticJavaResolveExtension source code*. <https://github.com/JetBrains/kotlin/blob/master/compiler/frontend.java/src/org/jetbrains/kotlin/resolve/jvm/extensions/SyntheticJavaResolveExtension.kt>
 - [23] JetBrains. 2023. *All-open compiler plugin*. <https://kotlinlang.org/docs/all-open-plugin.html>
 - [24] JetBrains. 2023. *AllOpen compiler plugin source code*. <https://github.com/JetBrains/kotlin/tree/master/plugins/allopen>
 - [25] JetBrains. 2023. *Kotlin Annotation Processing Tool*. <https://kotlinlang.org/docs/kapt.html>
 - [26] JetBrains. 2023. *Kotlin: Properties - Getters and Setters*. <https://kotlinlang.org/docs/properties.html#getters-and-setters>
 - [27] JetBrains. 2023. *Kotlin Symbol Processing API*. <https://kotlinlang.org/docs/ksp-overview.html>
 - [28] JetBrains. 2023. *Kotlinx serialization source code*. <https://github.com/JetBrains/kotlin/tree/master/plugins/kotlinx-serialization>
 - [29] JetBrains. 2023. *KSP: Limitations*. <https://kotlinlang.org/docs/ksp-why-ksp.html#limitations>
 - [30] JetBrains. 2023. *No-arg compiler plugin*. <https://kotlinlang.org/docs/no-arg-plugin.html>
 - [31] JetBrains. 2023. *NoArg compiler plugin source code*. <https://github.com/JetBrains/kotlin/tree/master/plugins/noarg>
 - [32] Oracle Joe Darcy. 2011. *JEP 117: Remove the Annotation-Processing Tool (apt)*. <https://openjdk.org/jeps/117>
 - [33] JUnit Team. 2023. *JUnit 5*. <https://junit.org/junit5/>
 - [34] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming, ECOOP 2011 (Lecture Notes in Computer Science, Vol. 2072)*. Springer, 327–353. https://doi.org/10.1007/3-540-45337-7_18
 - [35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. Getting started with ASPECTJ. *Commun. ACM* 44, 10 (2001), 59–65. <https://doi.org/10.1145/383845.383858>
 - [36] ktarger. 2023. *ktarger compiler plugin*. <https://github.com/ktarger/ktarger>
 - [37] Michael Ley. 2002. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002 (Lecture Notes in Computer Science, Vol. 2476)*, Alberto H. F. Laender and Arlindo L. Oliveira (Eds.). Springer, 1–10. https://doi.org/10.1007/3-540-45735-6_1
 - [38] Panagiotis Louridas. 2005. JUnit: Unit Testing and Coding in Tandem. *IEEE Softw.* 22, 4 (2005), 12–15. <https://doi.org/10.1109/MS.2005.100>
 - [39] MapStruct. 2023. *MapStruct annotation processor source code*. <https://github.com/mapstruct/mapstruct/tree/main>
 - [40] Mostafa Mehrabi, Nasser Giacaman, and Oliver Sinnen. 2018. 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, 405–414. <https://doi.org/10.1109/IPDPSW.2018.00075>
 - [41] Gabriel Menezes, Bruno B. P. Cafeo, and André C. Hora. 2022. How are framework code samples maintained and used by developers? The case of Android and Spring Boot. *J. Syst. Softw.* 185 (2022), 111146. <https://doi.org/10.1016/j.jss.2021.111146>
 - [42] Lukas Morawietz. 2022. *An Introduction to Kotlin Symbol Processing*. <https://www.codecentric.de/wissens-hub/blog/kotlin-symbol-processing-introduction>
 - [43] Brian Norman. 2020. *Writing Your Second Kotlin Compiler Plugin*. <https://bnorm.medium.com/writing-your-second-kotlin-compiler-plugin-part-1-project-setup-7b05c7d93f6c>
 - [44] Brian Norman. 2023. *kotlin-power-assert compiler plugin*. <https://github.com/bnorm/kotlin-power-assert>
 - [45] Oracle. 2023. *Compiler Tree API*. <https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/index.html>
 - [46] Oracle. 2023. *JavaDoc: Interface ProcessingEnvironment*. <https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javac/annotation/processing/ProcessingEnvironment.html>
 - [47] Renaud Pawlak. 2006. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distributed Syst. Online* 7, 11 (2006). <https://doi.org/10.1109/MDSO.2006.67>
 - [48] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Softw. Pract. Exp.* 46, 9 (2016), 1155–1179. <https://doi.org/10.1002/spe.2346>
 - [49] Project Lombok. 2023. *Project Lombok*. <https://projectlombok.org/>
 - [50] Viera K. Proulx and Weston Jossey. 2009. 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, 49–56. <https://doi.org/10.1145/1596655.1596663>
 - [51] Mark E. Royer and Sudarshan S. Chawathe. 2018. Java unit annotations for units-of-measurement error prevention. In *IEEE 8th Annual Computing and Communication Workshop and Conference, CCWC 2018*. IEEE, 816–822. <https://doi.org/10.1109/CCWC.2018.8301759>
 - [52] Myoungkyu Song and Eli Tilevich. 2012. Metadata invariants: Checking and inferring metadata coding conventions. In *34th International Conference on Software Engineering, ICSE 2012*. IEEE Computer Society, 694–704. <https://doi.org/10.1109/ICSE.2012.6227148>
 - [53] Myoungkyu Song and Eli Tilevich. 2015. Reusing metadata across components, applications, and languages. *Sci. Comput. Program.* 98 (2015), 617–644. <https://doi.org/10.1016/j.scico.2014.09.002>
 - [54] Myoungkyu Song, Eli Tilevich, and Wesley Tansey. 2009. 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, 813–814. <https://doi.org/10.1145/1639950.1640028>
 - [55] Egor Spirin, Egor Bogomolov, Vladimir Kovalenko, and Timofey Bryksin. 2021. PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 13–17. <https://doi.org/10.1109/MSR52588.2021.00014>
 - [56] Square Open Source. 2023. *KotlinPoet*. <https://square.github.io/kotlinpoet/>
 - [57] Ji Sunghin. 2023. *Say Hello to the Kotlin Compiler Plugin*. <https://betterprogramming.pub/say-hello-to-kotlin-compiler-plugin-f4e857be9a1>
 - [58] Vladislav Tankov, Yaroslav Golubev, and Timofey Bryksin. 2019. Kotless: A Serverless Framework for Kotlin. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. IEEE, 1110–1113. <https://doi.org/10.1109/ASE.2019.00114>

- [59] Wesley Tansey and Eli Tilevich. 2008. 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, 295–312. <https://doi.org/10.1145/1449764.1449788>
- [60] Eli Tilevich and Myoungkyu Song. 2010. Reusable enterprise metadata with pattern-based structural expressions. In *9th International Conference on Aspect-Oriented Software Development, AOSD 2010*. ACM, 25–36. <https://doi.org/10.1145/1739230.1739234>
- [61] Catalin Tudose and Carmen Odubasteanu. 2021. Object-relational Mapping Using JPA, Hibernate and Spring Data JPA. In *23rd International Conference on Control Systems and Computer Science, CSCS 2021*. IEEE, 424–431. <https://doi.org/10.1109/CSCS52396.2021.00076>
- [62] Uber. 2019. *Measuring Kotlin Build Performance at Uber*. <https://www.uber.com/blog/measuring-kotlin-build-performance/>
- [63] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. 2021. Characterizing the Usage, Evolution and Impact of Java Annotations in Practice. *IEEE Trans. Software Eng.* 47, 5 (2021), 969–986. <https://doi.org/10.1109/TSE.2019.2910516>
- [64] Wang Zaijun. 2022. *Do You Know the Principle of Lombok That Has Been Used for a Long Time?* <https://www.alibabacloud.com/blog/599443>

Received 2023-06-29; accepted 2023-07-31