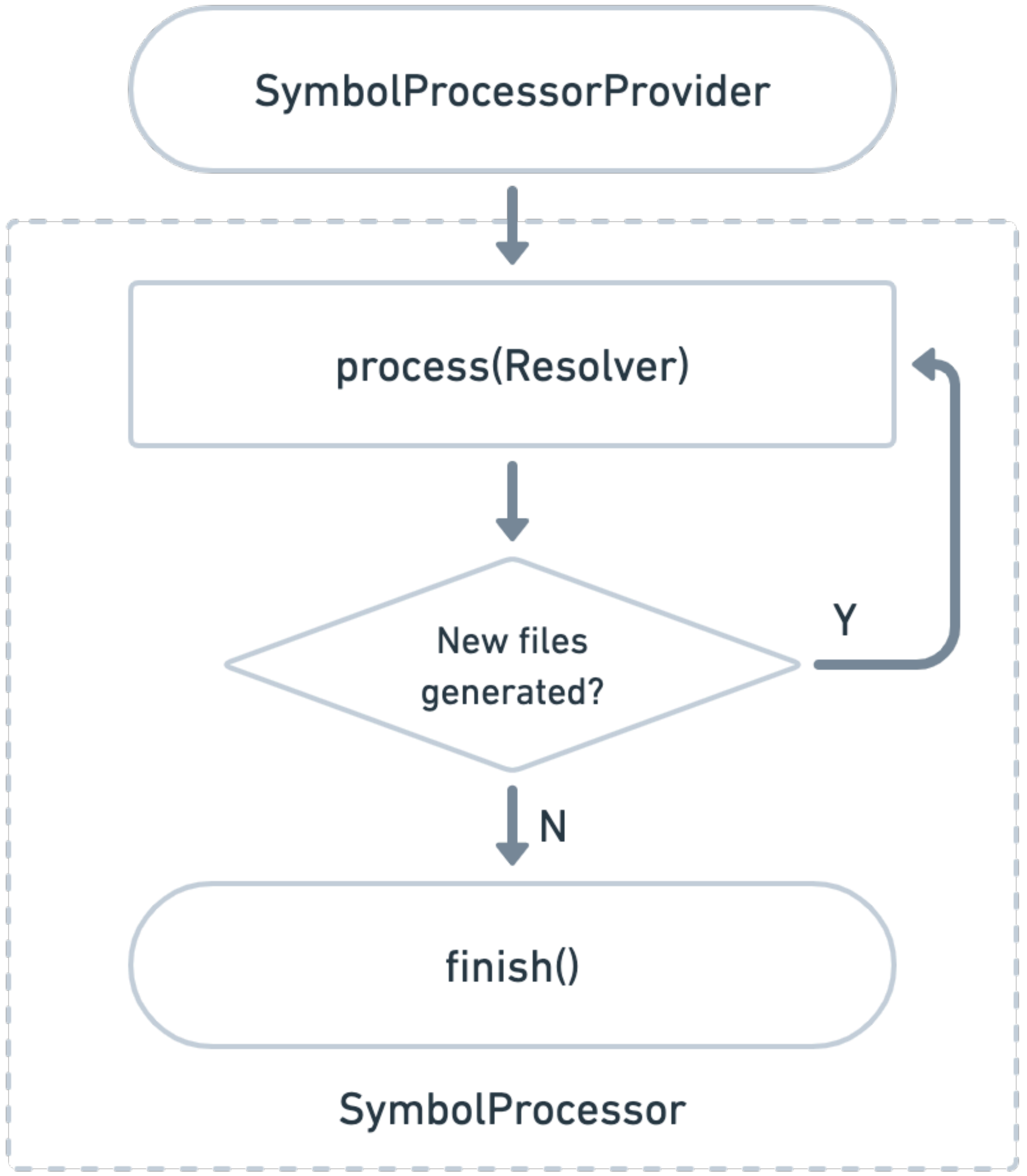
# **K**otlin **S**ymbol **P**rocessing *Tricks*

El(BingQuan) Zhang, Sr. Software Engineer

July 2022

KSP is a meta programming tool,
it leverages the Kotlin Compiler Plugin capability,
to export Kotlin Symbol information.

```
KSFile
  packageName: KSName
  fileName: String
  annotations: List<KSAnnotation>  (File annotations)
  declarations: List<KSDeclaration>
    KSClassDeclaration // class, interface, object
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      classKind: ClassKind
      primaryConstructor: KSFunctionDeclaration
      superTypes: List<KSTypeReference>
      declarations: List<KSDeclaration>
    KSFunctionDeclaration // top level function
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      …
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      …
```



New files generated by KSP are usually templates, helper-classes, etc.

## A sample with Room

```kotlin
@Dao
interface WordDao {

    // The flow always holds/caches latest version of data. Notifies its observers when the
    // data has changed.
    @Query("SELECT * FROM word_table ORDER BY word ASC")
    fun getAlphabetizedWords(): Flow<List<Word>>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(word: Word)

    @Query("DELETE FROM word_table")
    suspend fun deleteAll()

}
```

```java
@Override
public Flow<List<Word>> getAlphabetizedWords() {
  final String _sql = "SELECT * FROM word_table ORDER BY word ASC";
  final RoomSQLiteQuery _statement = RoomSQLiteQuery.acquire(_sql, 0);
  return CoroutinesRoom.createFlow(__db, false, new String[]{"word_table"},
    new Callable<List<Word>>() {
    @Override
    public List<Word> call() throws Exception {
      final Cursor _cursor = DBUtil.query(__db, _statement, false, null);
      try {
        final int _cursorIndexOfWord = CursorUtil.getColumnIndexOrThrow(_cursor, "word");
        final List<Word> _result = new ArrayList<Word>(_cursor.getCount());
        while(_cursor.moveToNext()) {
          ...
        }
        return _result;
      } finally { ... }
    }

    @Override
    protected void finalize() {
      _statement.release();
    }
  });
}
```

## KSP is 2x faster than KAPT.

KAPT has been moved to the maintenance mode.

## Go DEEP

Work with **Custom Symbol Filters**
Work with **Kotlin Dependencies**
Work with **Multi Processors**

## Go BROAD

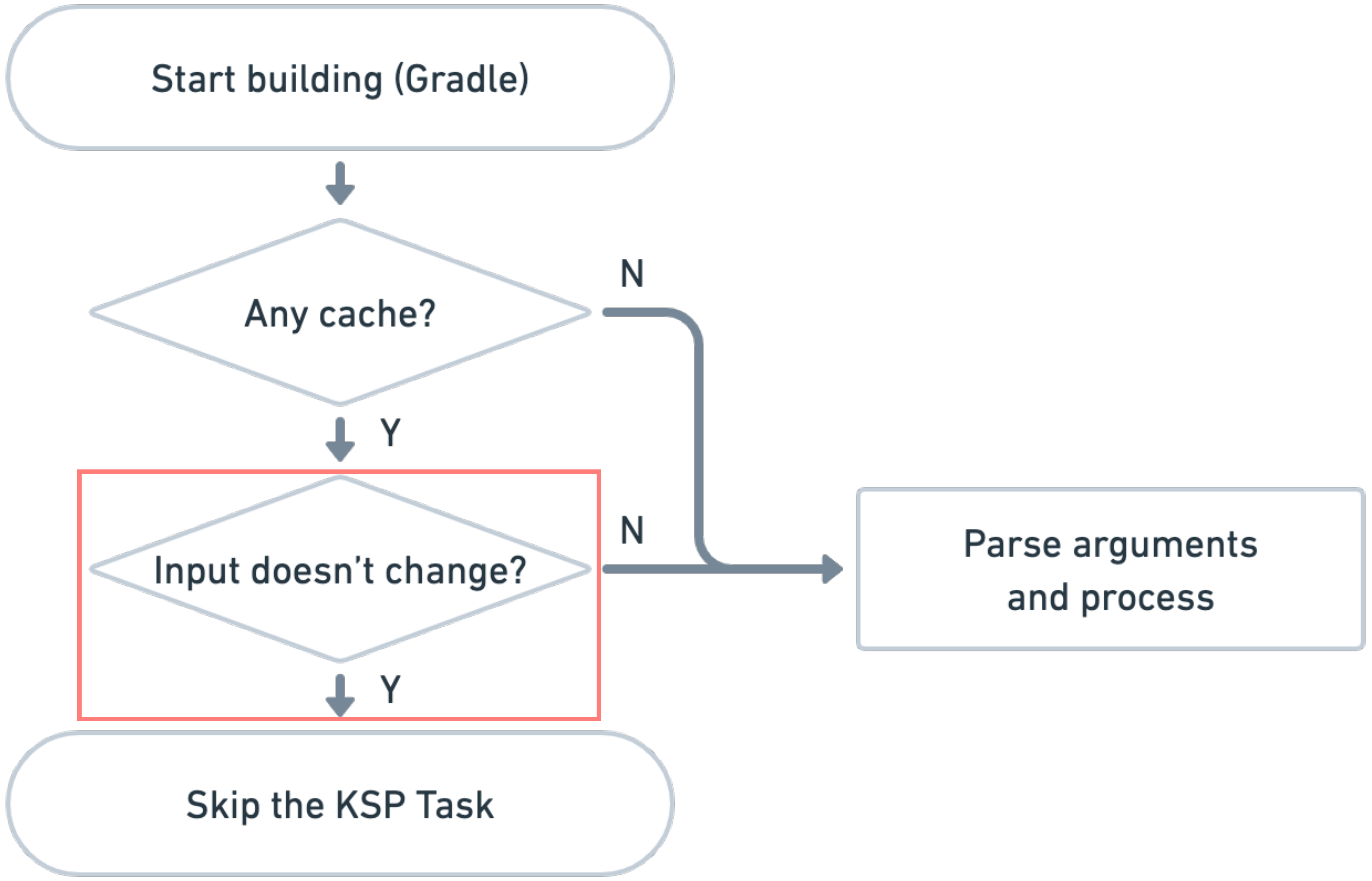
Work with **Gradle**
Work with **Variant**

## 1. File Arguments

When passing complex arguments to the KSP processor, it’s unsafe for just using the regular key-value pair API.

```
ksp {
  arg("procA.input", "in.json")
  arg("procB.config", "in.xml")
  ...
}

val path = symbolProcessorEnvironment
    .options[“procB.config”]

File(path).readText()
```

KSP has no idea on how to distinguish whether the value is a File or not.


Start building (Gradle)
Any cache?
N
Y
Input doesn't change?
N
Y
Parse arguments and process
Skip the KSP Task

If the content of the target File changed, KSP task need to be aware of that.

## Task Runtime API

```
project.tasks.withType<com.google.devtools.ksp.gradle.KspTask> {
    inputs.files(project.layout.projectDirectory
        .file("in.json")
    )
}
```

## The CommandLineArgumentProvider

```kotlin
class InOutFilesProvider(
    @InputFile
    @PathSensitive(PathSensitivity.RELATIVE)
    val localProperties: File
) : CommandLineArgumentProvider {
    override fun asArguments(): Iterable<String> {
        return listOf("myProc.localProps=${localProperties.path}")
    }
}
...
ksp {
    arg(InOutFilesProvider(rootProject.layout
        .projectDirectory.file("local.properties").asFile))
}
```

https://github.com/google/ksp/pull/872/files

## How does it work?

```kotlin
interface KspTask : Task {
    @get:Nested
    val commandLineArgumentProviders: ListProperty<CommandLineArgumentProvider>
    ...
}
```

However, when using Android Gradle plugin 3.2.0 and higher, you need to pass processor arguments that represent files or directories using Gradle's `CommandLineArgumentProvider` interface.

Using `CommandLineArgumentProvider` allows you or the annotation processor author to improve the correctness and performance of incremental and cached clean builds by applying incremental build property type annotations to each argument.

For example, the class below implements `CommandLineArgumentProvider` and annotates each argument for the processor. The sample also uses the Groovy language syntax and is included directly in the module's `build.gradle` file.

*https://developer.android.com/studio/build/dependencies*

## 2. Variant-aware Arguments

Sometimes we want a feature only enabled in debug mode. How to pass variant-aware args (e.g. with AGP variants mechanism)?

```
val isDebug = gradle.startParameter
    .taskRequests.toString()
    .contains("debug").toString()
ksp {
    arg("logEnabled", isDebug)
}

// However above snippet doesn’t work
// for `./gradlew assemble` command
```

## Config respectively

```
ksp {
    arg("logEnabled.debug", "true")
    arg("logEnabled.preRelease", "false")
    arg("logEnabled.release", "false")
}

// buildTypes {
//     getByName("debug") {
//         isDebuggable = true
//     }
//     getByName("release") {
//         isDebuggable = true
//     }
// }
// Either
env.options["logEnabled.debug"]
// Or
env.options["logEnabled.release"]
```

## Well... but how to extract them in processor?

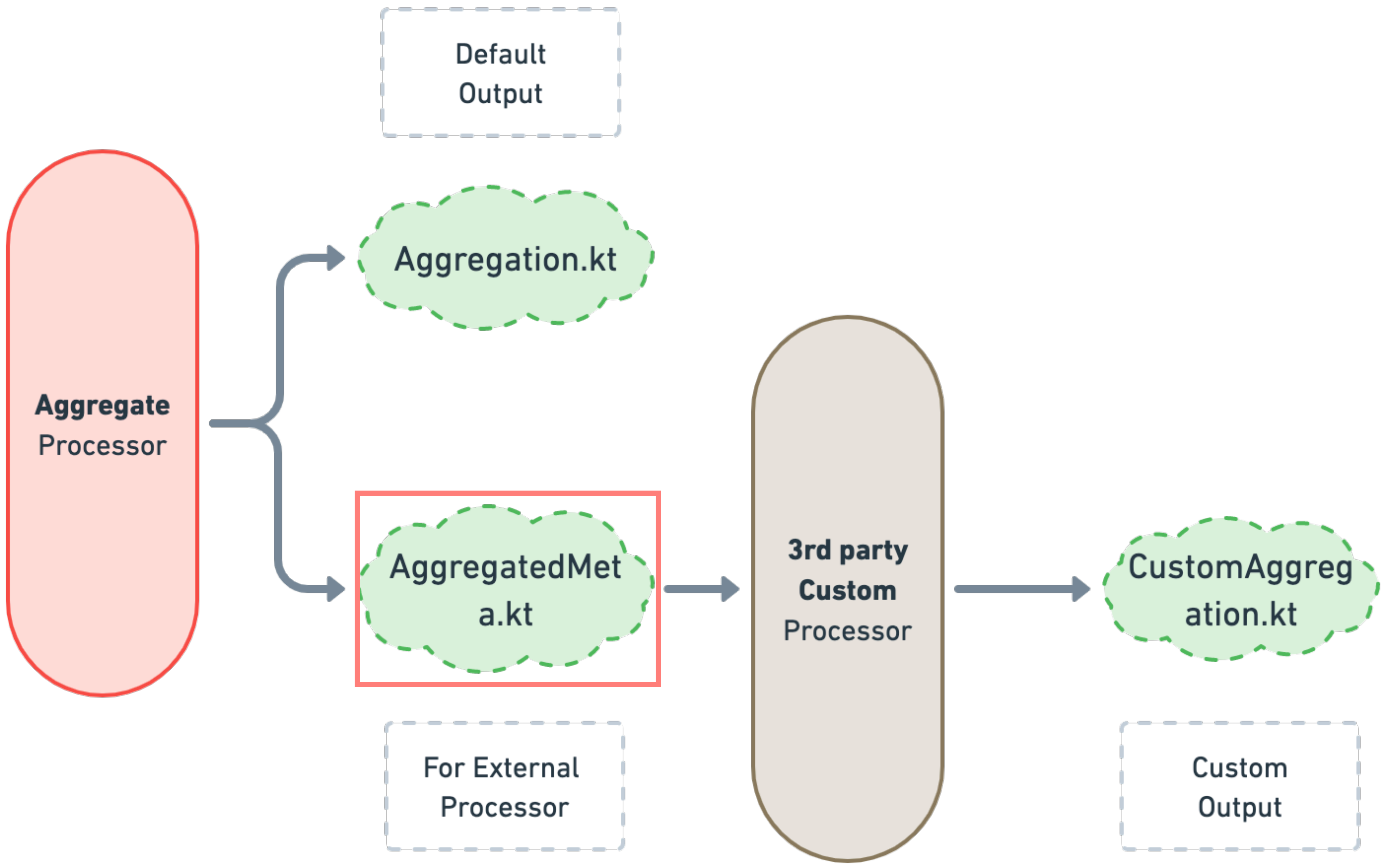
## Workaround (reflection)

```
class VariantAwareness(env: SymbolProcessorEnvironment) {

    val variantName: String

    init {
        val resDir = CodeGeneratorImpl::class.memberProperties
            .first { it.name == "resourcesDir" }
            .also { it.isAccessible = true }
            .getter(env.codeGenerator as CodeGeneratorImpl)
            .toString()
        variantName = File(resDir).parentFile.name
    }
    ...
}
// env.options["logEnabled.${variantAwareness.variantName}"]
```

# 3. Multi-Processors

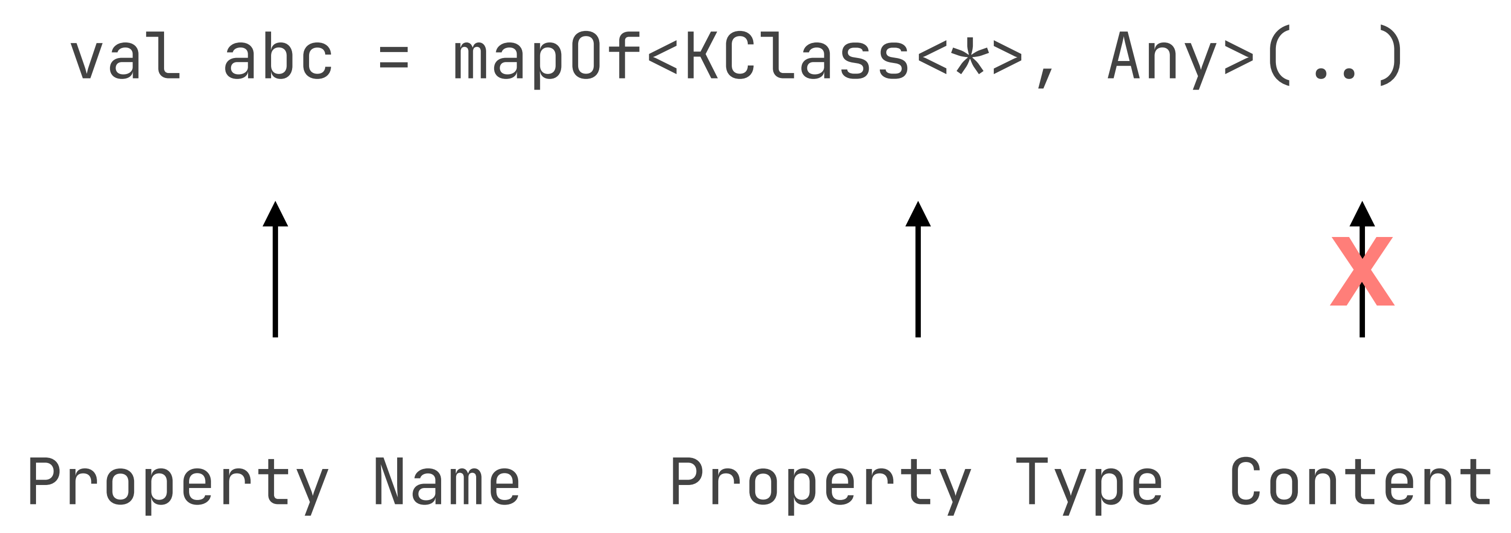
A processor that exports aggregation source code/report for external users, may receive requests from user on customization options.

```
// Assume our processor generates Aggregation.kt
// as a final file for external users to use.
val annotatedClasses = mapOf<KClass<out Annotation>,
        List<ClassDeclarationRecord>>(..)



// However user may not want the structure like above…
// the best way is to offer raw data and let user decide
// the generation format/structure
```



```
@Extend
Elements…  ⟶  getSymbolsWithAnnotation()
```

# No Expression.
# Symbol/Declaration Only.



```
val abc = mapOf<KClass<*>, Any>(..)
```

Exception: Annotation parameters are the only resolvable data.

# Aggregated metadata

```
// AggregatedMeta.kt
@Extend(metaDataInJson = """{
   "annotatedClasses":{
      "examaple.ExportActivity":[
         {
            "name":"me.xx2bab.koncat.sample.android
                        .AndroidLibraryActivity”,
            "Annotations":[ … ]
         },
         …
      ]
   },
   "typedClasses":{ … },
   "typedProperties":{ … }
}
""")
val voidProp = null // DO NOT use voidProp
```

## 4. Dependencies

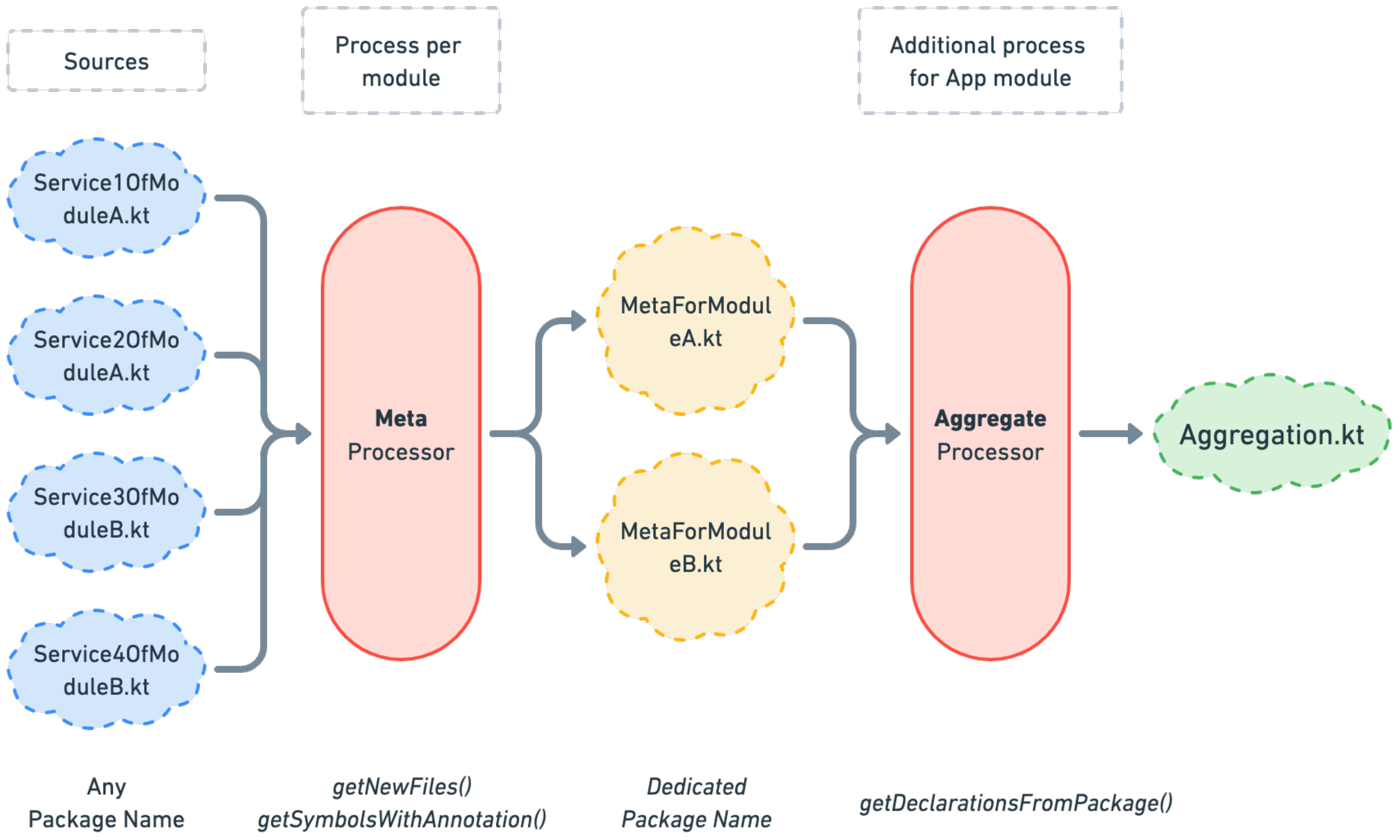
Usually, the processing scope limits to source files of a single module. While aggregating symbols from multi-modules is also a common case, is KSP capable of supporting it?

```
// Assume we would like to generate
// the code below for app module

// Application
fun onCreate() {
     registerServices(
          ServiceFromModuleA(),
          ServiceFromModuleB()
     )
}

// Other modules
@ExportService
Class ServiceFromModuleA
@ExportService
Class ServiceFromModuleB
```

## Retrieve declaration from deps

```
interface Resolver {
    fun getKSNameFromString(name: String): KSName

    fun getClassDeclarationByName(name: KSName): KSClassDeclaration?

    fun getFunctionDeclarationsByName(name: KSName, includeTopLevel: Boolean = false):
Sequence<KSFunctionDeclaration>

    fun getPropertyDeclarationByName(name: KSName, includeTopLevel: Boolean = false):
KSPropertyDeclaration?

    fun getDeclarationsFromPackage(packageName: String): Sequence<KSDeclaration>

    ..
}
```

*https://github.com/google/ksp/issues/344*


Sources
Process per module
Additional process for App module
Service1OfModuleA.kt
Service2OfModuleA.kt
Service3OfModuleB.kt
Service4OfModuleB.kt
**Meta** Processor
MetaForModuleA.kt
MetaForModuleB.kt
**Aggregate** Processor
Aggregation.kt
Any Package Name
*getNewFiles()*
*getSymbolsWithAnnotation()*
*Dedicated Package Name*
*getDeclarationsFromPackage()*

## Metadata of a module

```
// MetaFor${ModuleName}.kt
@Meta(metaDataInJson = """{
    "annotatedClasses":{
        "examaple.ExportActivity":[
            {
                "name":"me.xx2bab.koncat.sample.android
                                .AndroidLibraryActivity”,
                "Annotations":[ … ]
            },
            …
        ]
    },
    "typedClasses":{ … },
    "typedProperties":{ … }
}
""")
val voidProp = null // DO NOT use voidProp directly
```

## Final Aggregation

```kotlin
// Aggregation.kt
val annotatedClasses = mapOf<KClass<out Annotation>,
          List<ClassDeclarationRecord>>(..)
val interfaceImplementations = mapOf<KClass<*>, Any>(..)
val typedProperties = mapOf<KClass<*>, Any>(..)
```

## 5. Beyond AP

One of the KSP core functionality is to transform source code to a symbol tree. Thus, actually you can do more than just Annotation Processing.

```
KSFile
  packageName: KSName
  fileName: String
  annotations: List<KSAnnotation>  (File annotations)
  declarations: List<KSDeclaration>
    KSClassDeclaration // class, interface, object
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      classKind: ClassKind
      primaryConstructor: KSFunctionDeclaration
      superTypes: List<KSTypeReference>
      declarations: List<KSDeclaration>
    KSFunctionDeclaration // top level function
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      …
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      …
```

## getSymbolsWithAnnotation()

```
class SampleProcessor(...) : SymbolProcessor {
    override fun process(resolver: Resolver): List<KSAnnotated> {
        resolver.getSymbolsWithAnnotation("com.example.Anno")
            .forEach { ksAnnotated: KSAnnotated →
                ...
            }
        }
}

// 1. Fetch new source files of this round
//    (regarding to multi-round processing)
// 2. Create a visitor to filter elements by specific Annotation
```

## Explore more via getNewFiles()

```
class SampleProcessor(...) : SymbolProcessor {
    override fun process(resolver: Resolver): List<KSAnnotated> {
//          resolver.getSymbolsWithAnnotation("com.example.Anno")
//              .forEach { ksAnnotated: KSAnnotated ->
//
//          }
        resolver.getNewFiles().forEach { ksFile: KSFile ->
            ksFile.accept(visitor, data)
        }
        return emptyList()
    }
}
```

## Filter by class types

```kotlin
override fun visitClassDeclaration(classDeclaration: KSClassDeclaration) {
    val className = classDeclaration.qualifiedName?.asString()
    if (className.isNullOrBlank()) { // Anonymous class is not supported
        return
    }
    classDeclaration.superTypes.forEach { superType ->
        val superKSType = superType.resolve()
        if (superKSType.toClassName().canonicalName == "kotlin.Any") {
            return@forEach
        }
        targetClassTypes.forEach { targetClassType ->
            if (superKSType.isAssignableFrom(targetClassType.type)) {
                ...
            }
        }
    }
}
```

## Filter by class types

```
// Application
fun onCreate() {
    registerServices(
        ServiceFromModuleA(),
        ServiceFromModuleB()
    )
}

// Other modules
@ExportService
Class ServiceFromModuleA
@ExportService
Class ServiceFromModuleB
```

```
// Application
fun onCreate() {
    registerServices(
        ServiceFromModuleA(),
        ServiceFromModuleB()
    )
}

// Other modules
// @ExportService
Class ServiceFromModuleA: ServiceFlag
// @ExportService
Class ServiceFromModuleB: ServiceFlag
```

# Koncat

**Aggregate Kotlin Symbols based on KSP for multi-modules development in compile-time.** For instance, when you want to gather all implementations of an interface across multi-modules, Koncat must be the tool your shouldn't miss.

**https://github.com/2BAB/Koncat**

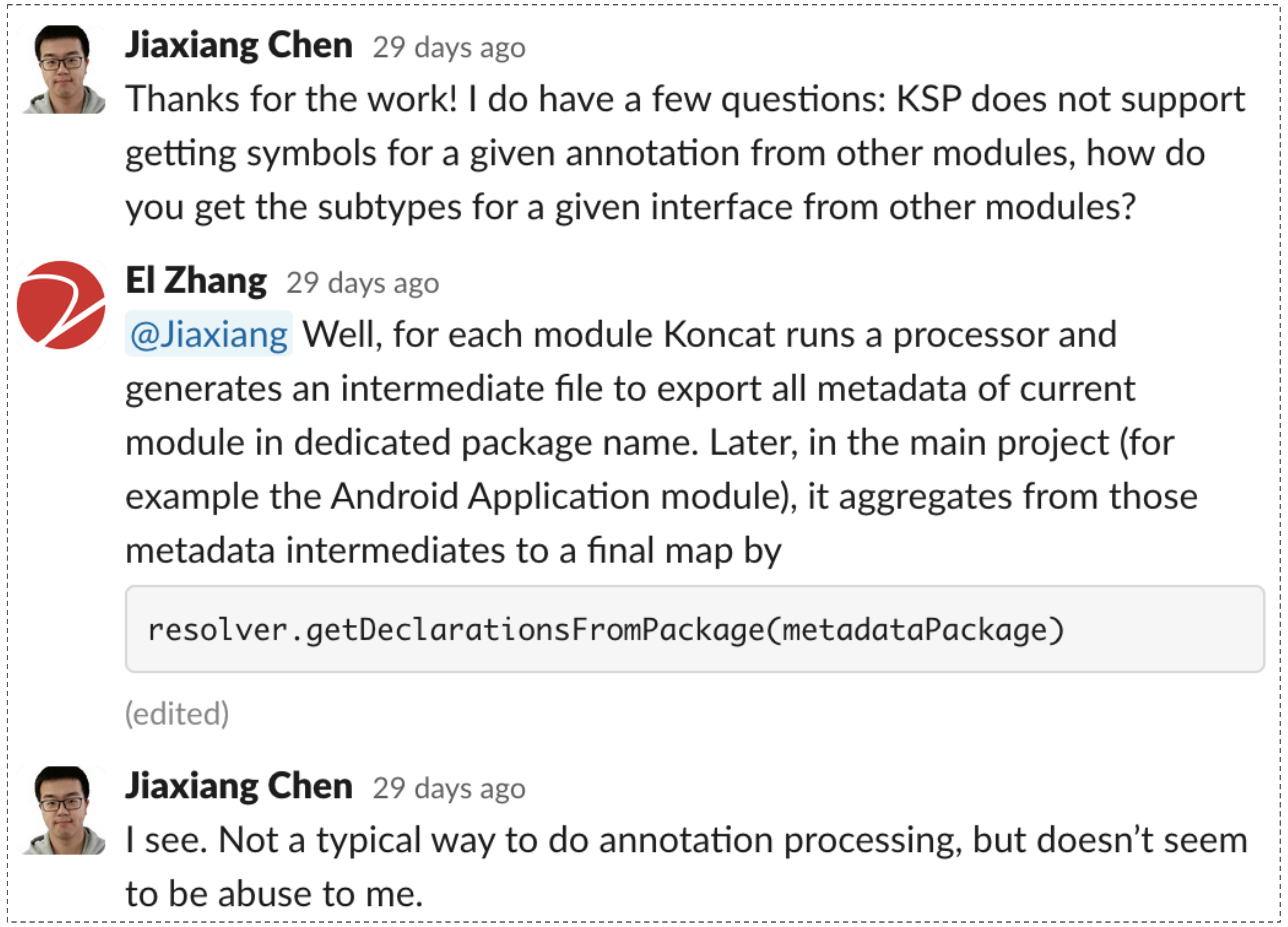
```
// build.gradle.kts for application module
koncat {
    annotations.addAll("sample.annotation.ExportActivity")
    classTypes.addAll("sample.interfaze.DummyAPI")
    propertyTypes.addAll("org.koin.core.module.Module")
}

// Aggregation.kt
val annotatedClasses = mapOf<KClass<out Annotation>,
    List<ClassDeclarationRecord>>(..)
val interfaceImplementations = mapOf<KClass<*>, Any>(..)
val typedProperties = mapOf<KClass<*>, Any>(..)
```

## Koncat

**Aggregate Kotlin Symbols based on KSP for multi-modules development in compile-time.**

For instance, when you want to gather all implementations of an interface across multi-modules, Koncat must be the tool your shouldn't miss.

https://github.com/2BAB/Koncat



**Jiaxiang Chen** 29 days ago
Thanks for the work! I do have a few questions: KSP does not support getting symbols for a given annotation from other modules, how do you get the subtypes for a given interface from other modules?

**El Zhang** 29 days ago
@Jiaxiang Well, for each module Koncat runs a processor and generates an intermediate file to export all metadata of current module in dedicated package name. Later, in the main project (for example the Android Application module), it aggregates from those metadata intermediates to a final map by

```
resolver.getDeclarationsFromPackage(metadataPackage)
```

(edited)

**Jiaxiang Chen** 29 days ago
I see. Not a typical way to do annotation processing, but doesn't seem to be abuse to me.

# Thank you!

xx2bab@gmail.com

github.com/2BAB

twitter.com/xx2bab