JETBRAINS
DART HINTS PLUGIN

MOHAMMAD SALEH AMIRI

ABSTRACT

The project idea is to add information to the code editor while coding in Dart programming language. The project aims to assist coders in understanding code more quickly while writing and reading it. This project intended to extend the functionality of an integrated development environment (IDE) called intellij idea version 2021.2.4 Through a plugin supports Dart programming language.

This plugin will support dart programming language developers through the information provided. The information that will be provided are static variables types, methods parameters names and types. The plugin will add them to the editor in real time where the types and names are not explicitly mentioned. like where (var) keyword is used in definition, the variable type is not explicitly mentioned, and in method calls.

The plugin used proper set of tools:

- intellij idea source code mainly. Not whole source code was used directly. only 3 packages were used: Com.intellij.codeinsight, Com.intellij.openapi, Com.intellij.psi.
- Dart Program Structure Interface (Dart PSI).
- Gradle.

The plugin development resulted in a toggle button in editor popup menu -Right Click on Editor-. When a user toggles the button, hints get generated, and as user writes code, more hints are generated if required, the user could toggle off the feature by simply clicking the button again.

Contents

Figures

TERMS

PSI:	**Program Structure Interface.**
IDE:	**Integrated Development Environment.**
UI:	**User Interface.**
FFBD:	**Functional Flow Block Diagram.**
XML:	**Extensible Markup Language.**
UML:	**Unified Modeling Language.**

Section 1: Introduction to project idea

1.1 Introduction

In the early days of programming, it was a really hard job. Programmers used to write code inside a primeval environment (Text Editor), where you can’t tell whether a code have a syntax error or if you have written something wrong unless you compile the code. or you could get a headache reading every single line of code to see if there is something wrong. Some problems were not caused by the writer of the code, for example, using someone else’s code could be problematic if no documentation is available, all of these were problems facing ancient programmers. Since then, programmers started to develop better text editors with many features, and named them Integrated Development Environment (IDE). IDEs made the life of a programmer a lot easier with their features, Such as colorful syntax, syntax error highlighting, local documentations and many other features that made logical errors the only errors that could be unnoticed while writing a code.

One of these features is called (Inlay Hints), it was first developed and introduced in 2016 for (Intellij Idea) IDE by the software company JetBrains. This feature is a tool that was developed only for java programming language. it is used in real time code editing to view specific information like variables types and method call parameters names, and few other similar information. In time, The company supported this feature to other programming languages like kotlin and groovy and typescript and some other languages.

this report will discuss an implementation of a feature called (Dart Inlay Hints) added to Intellij Idea IDE as a plugin, this feature adds hints to (Dart) programing language when editing a dart file.

This section will discuss (Hints) feature’s purpose, importance, tools used in the process of developing Dart Inlay Hints and the writing of this report. Also background studies will be discussed as there are other similar works to this work.

1.2 Project Problem

The problem of this project lies with the fact that as the code gets bigger as it becomes harder to remember and recognize. For example, if you have this method call: Method40(_,_,_,_,_); , it is very easy to forget what arguments were required for this specific method. Also when we have a long chain of calls with a returned type, it could be easy to not recognize the returned type immediately. This project manages to presents a solution for this problem for **Dart** programming language.

1.3 Project Purpose and importance

The project purpose is to add a feature called (Dart Inlay Hints) to intellij idea 2021.2.4 IDE as a (Plugin[1]). It is used to assist the writer of dart code by eliminating the type ambiguity [2]from the static code through adding variable and method call parameters hints.

1.4 Why Intellij Idea?

This tool was developed for intellij idea platform mainly because the developers of this plugin prefer intellij idea over other IDEs. Other reason is because intellij idea already has an infrastructure that supports third party plugins and already has an infrastructure that supports inline editor words that are ignored by the compiler just like comments, in the case of this project they are (Hints).

1.5 Background Studies

In-editor Hints are not new. as said, JetBrains developed this feature for it's platform (intellij idea) in 2016 for Java programming language. It supports variable and parameter name hints and Java Annotation [3]and chain call hints[4].

an example of Variable and Parameter name hints:

var ***StaticTypeHint*** variableName = Method1(***paramName1:*** 5, ***paramName2:*** "Moh");

StaticTypeHint: it is the hint of the actual static returned type from (Method1) that will be assigned to **variableName**.

paramName1,2: these are the hints which represents the name of the parameter in that position in the method signature.

After JetBrains invented Inlay Hints feature for java in Intellij Idea, many other languages got supported with the same feature across all jetbrains products, C# in JetBrains Rider, multiple web languages in Webstorm, C&C++ in Clion. Also other software companies, like Microsoft, decided to copy the feature to their platform, Visual Studio. it is named

[1] Plugin: a plugin is a software that is embedded to other larger software, and it represents a specific functionality in the larger software, and the plugin can't work by itself.

[2] Type ambiguity: it refers to the variables defined without explicitly mentioning the static type with definition. E.g. x =5; and var x =5;. It refers also to method parameters when calling, parameters are ambiguous.

[3] Java Annotation: annotations are the word starts with @ , such as @override, @Nullable, @NotNull.

[4] Chain call hints: Chain call hints are hints shown when we make a chain of calls and in some points the returned type changes. These is where it is shown.

(Inline Hints) in visual studio and it supports variable types and parameter name hints for C#.

1.6 Conclusion

This section has discussed the project idea and overview of tools and goal. as discussed, the project idea is to add a feature as a plugin to a platform called intellij idea, and this plugin purpose is to assist any coder that code in dart programming language through adding hints to the code editor in real time that represents the variable types and parameter types and names. Also the tools were discussed, this project used intellij idea source code, and dart programming language source code, and gradle build tool.

Section 2: Analysis of Tools Architecture

2.1 Introduction

In this section will discuss the crucial tools architecture used for development in details. What will be handled in this section is: tools used in development phase, Intellij Idea Architecture, PSI, Dart Source Code, Gradle. understanding this section will qualify enable the reader to understand the diagrams of the project in the next section.

2.2 Project Tools

The tools divided into 2 parts, the tools used to develop the featured tool, tools used to write this report.

2.2.1 Dart Inlay Hints Tools

- **Intellij Idea 2021.2.4 Source code**

Understanding the structure of intellij idea platform and plugins system is crucial part of understanding the development phase of this plugin. Intellij idea is a collection of plugins written in java and kotlin bundled together, each plugin has a specific task in the IDE, and all plugins work together to present the IDE as we know it. because of this structure, JetBrains was capable to present multiple domain specific IDEs (Intellij Idea, PyCharm, Rider, etc..) with almost identical structure, as it uses the same bundled plugins in all of it's products plus some domain specific plugins. the structure will be discussed deeply in the next section.

Intellij idea 2021.2.4 source code was required because the plugin is developed for it. As for sure, not the whole source code was used directly, few packages were used to develop this plugin:

- **Com.intellij.codeinsight**

This package is used in the platform to present various in-editor information, such as documents, syntax error highlight, coloring, hints, indentations, folding, and more [1]. For this project only hints functionalities were used (**com.intellij.codeinsight.hints.***) which is used to add inline text ignored by the compiler and has other multiple properties.

- **Com.intellij.openapi**

This package is used in various functionalities in the platform, editor features, abstract IO features, IDE actions (every button in the IDE represents an action) [2], It is major part of the IDE. Only editor and action features were used (**com.intellij.openapi.editor.***) which is used to access the editor and manipulate it, (**com.intellij.openapi.actionsSystem.***) which is used create a way for user to interact with the plugin behavior -a clickable button for example-.

- **Com.intellij.psi**

This package is the most important package in the whole platform. First of all, psi stands for “Program Structure Interface”. This package works as A layer between the language and all platform features, “it is responsible for parsing files and creating the syntactic and semantic code model that powers so many of the platform features” [3]. Every programming language must implement a PSI layer to communicate with the features available in the platform, jetbrains provides abstract PSI module that must be implemented by any language to make the platform supports it [4]. This project uses two different PSI layers, the first one is intellij idea implementation of PSI layer to communicate with all platform features, the second one is Dart language implementation of PSI layer to recognize Dart code and manipulate it. The libraries used are (**com.intellij.psi.***) which was used to manipulate intelij idea components as elements, files as this report will discuss later, (**com.jetbrains.lang.dart.psi.***).

- **Dart language Source code**

Dart source code is downloaded and added to the project to make the platform recognizes dart code as dart code using (Dart PSI layer) and add hints to dart code only.

- **Gradle Build Tool**

Gradle build tool [5]is a tool developed by Gradle incorporation. It is an open source build automation tool used to build any type of software (scripts), it is written in groovy and kotlin and java [5]. This build tool is used by intellij platform to automate build process of plugins and other tasks like generating json files for exporting plugins to internet, generating certificates for the plugins, adding project dependencies and many other tasks that have the same process across all project but different values [6].

2.2.2 Report Tools

- **Microsoft Office Visio**

Microsoft Office Visio is used to draw the diagrams of the project in this report.

2.3 Intellij Idea Platform Architecture

Intellij Idea is the major IDE for JetBrains, is an event driven[6] platform, it is built out of multiple separate plugins, each plugin represents a functionality, and all plugins developed

[5] Build Tool: build tools are tools used to automate any task with periodic process such as tasks of compilation to various extensions (exe,json,dll,etc..), packaging, testing, deployment, publishing.
[6] Event Driven: An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update.[7]

using java and kotlin, what makes Intellij Idea unique is that JetBrains managed to fragment its functionalities, thus making all of its parts reusable because each plugin is separate from the other, but they all work together. JetBrains worked on multiple IDE's descendent from intellij idea. WebStorm, Rider, CLion, PyCharm, etc.., they all were developed using almost the same tools. This section will discuss the backend part of the platform and how it works.

Intellij Idea internal parts works with each other through a layer called Program Structure Interface (PSI). everything in the IDE have to implement a class that extends PSI in a way. because of generalization (having the same infrastructure for everything), JetBrains was capable to copy and paste the same code for all of it's IDE's.

2.3.1 Program Structure Interface (PSI)

The Program Structure Interface, commonly referred to as just PSI, is the layer in the IntelliJ Platform responsible for parsing files and creating the syntactic and semantic code model that powers so many of the platform's features. [3]

The Program structure interface also used to implement the lexer and parser of any programming language supported on the platform [8]. It also consists of many interfaces with enormous amount of classes that implements them, each language implements it's own PSI layer that is descendent from the original PSI layer [9], thus it will be impossible to explain the whole structure of PSI, but a glance will be taken of what PSI is. each functionality in the IDE is represented by multiple PSI interfaces implemented for that specific functionality.

2.3.1.1 PSIElement

There is an interface called **PSIElement** (**com.intellij.psi.PsiElement**), this interface is a major interface that is implemented or extended directly or indirectly by all other interfaces that represents functionalities (features). For example, in order to make the platform recognize a file type, each language must implement an interface called **PSIFile** that in its turn inherit **PSIElement** [10]. The java file type for example is recognized through an implementation class (**com.intellij.psi.impl.PsiJavaFileImpl.java**) and Dart file type is recognized through an implementation class (**com.jetbrains.lang.dart.psi.DartFile.class**).

[Figure 2 – 1] and [Figure 2 - 2] are examples of the java file dependency tree [7]and dart file dependency tree generated using **JetBrains UML Class diagram tool**. The images represents how any file type must implement the PSIElement:

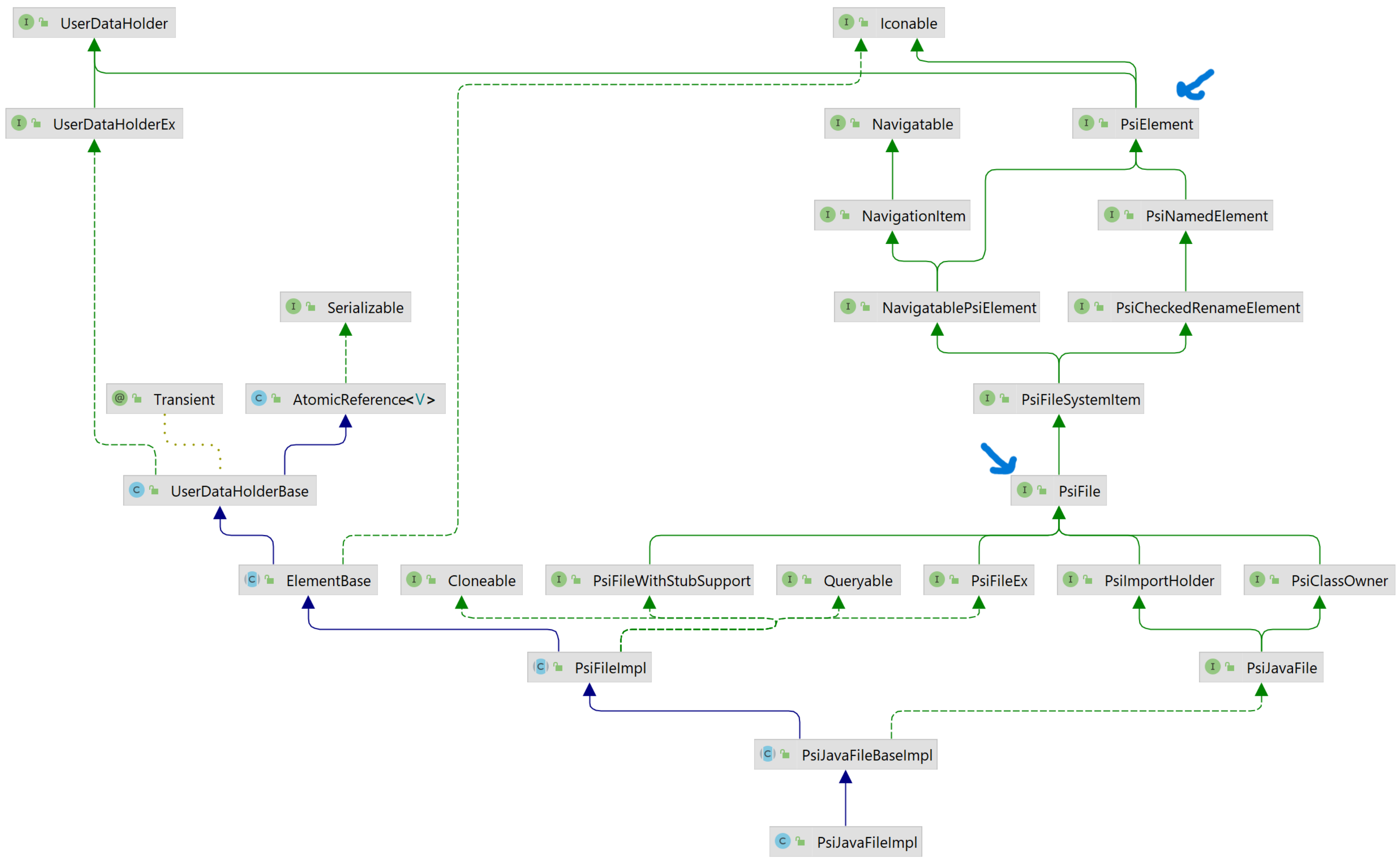


Figure 2 - 1: Java File Type Generated Dependency Tree

[7] Dependency Tree: it is a way to represent the hierarchy of classes inheritance and interfaces that are implemented of a class.

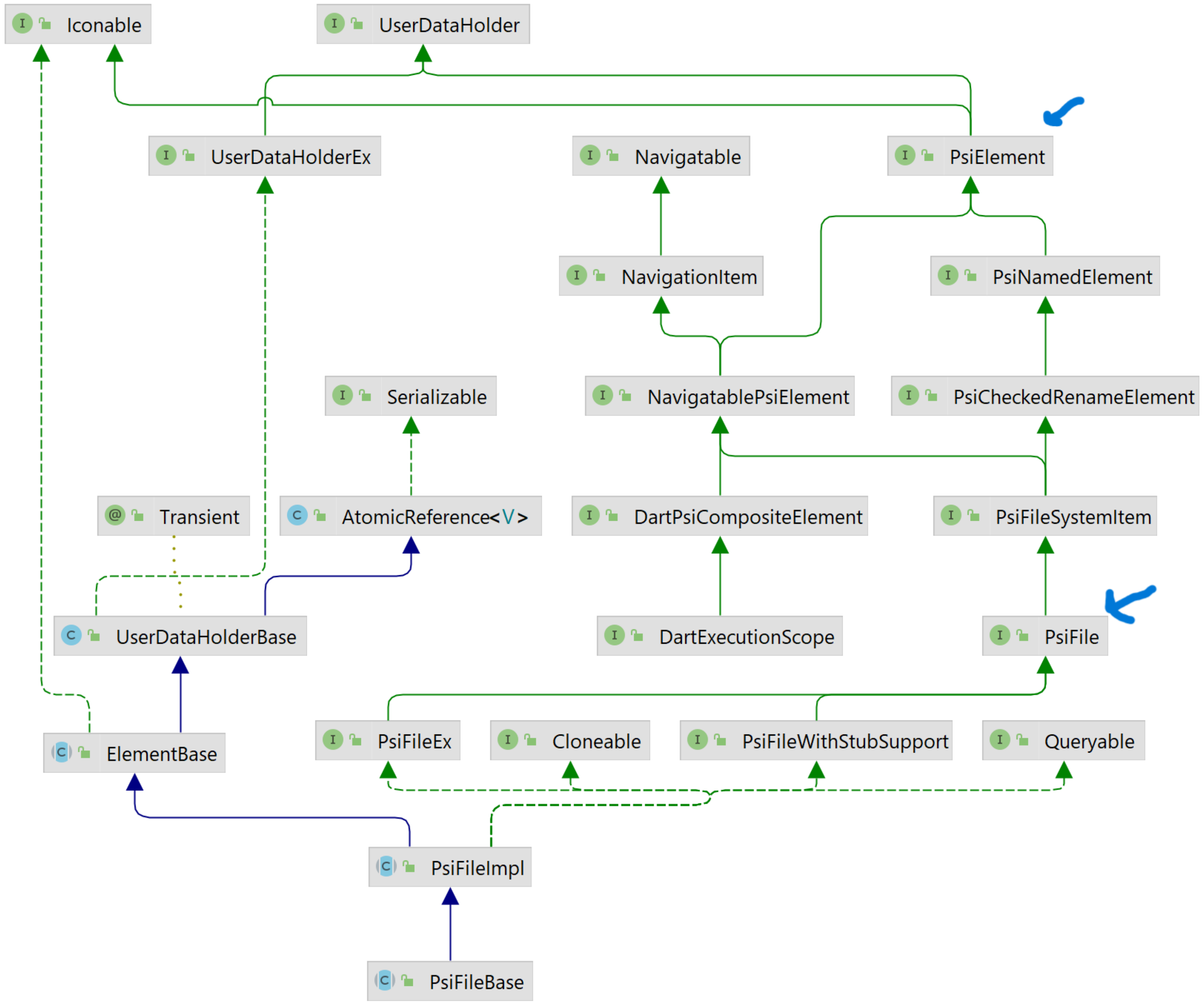


Figure 2 - 2: Dart File Generated Dependency Tree

through the PSI file (Which is a PSIElement too), we can manipulate everything in it, we can extract the code as text for example through **psiFile.getText();**. Also every single word in the code is considered a **PSIElement** indirectly because as said, the analysis phase [8]of compilation is implemented using PSI, so every variable and method and definition and expression etc.. can be accessed as an element that has some attributes (Fields).

The [Figure 2 – 3] shows how important the PSIElement Interface in Intellij idea platform; it is extended or implemented by more than 1500 class and interface across the platform, every

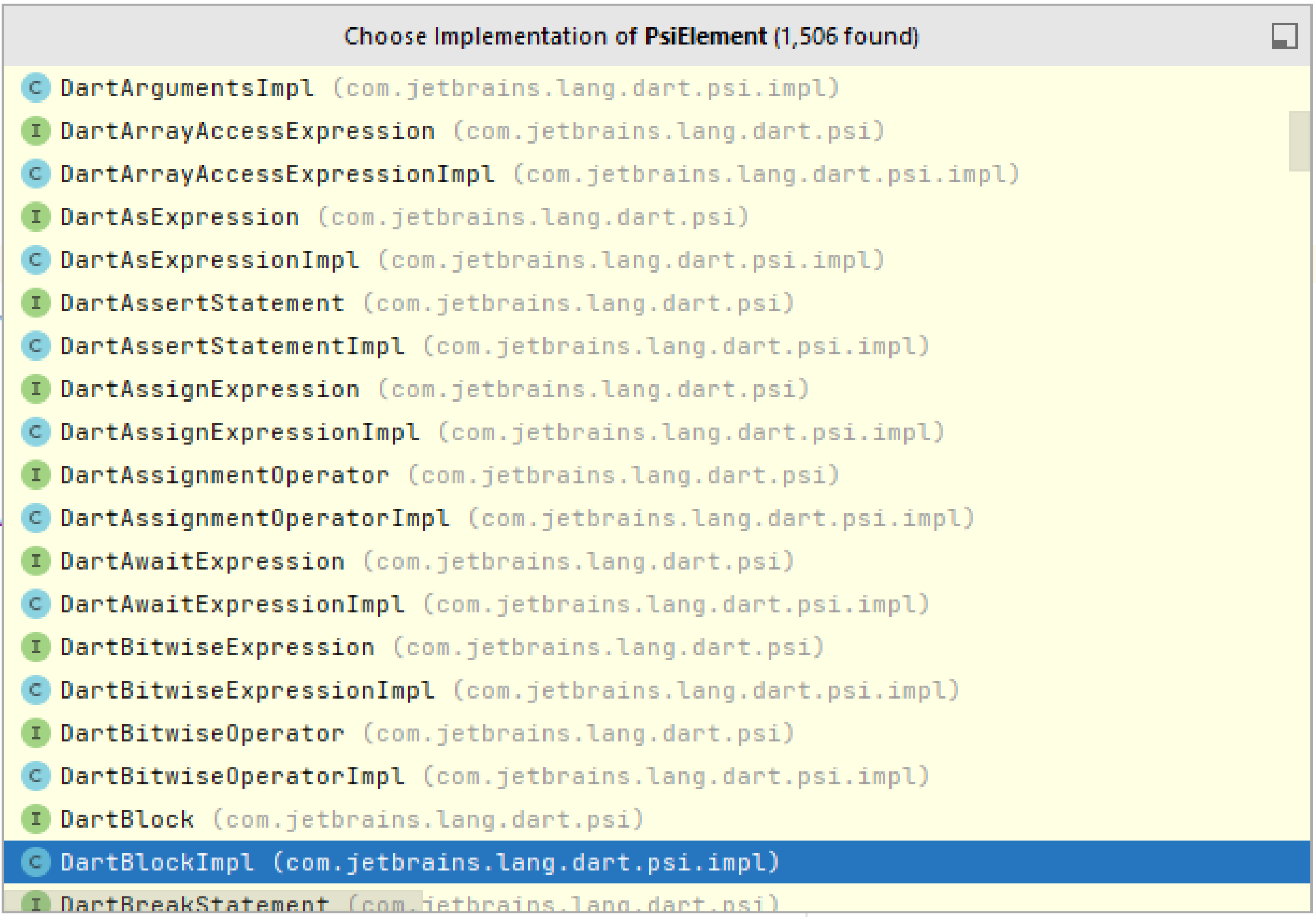

```
Choose Implementation of PsiElement (1,506 found)
C DartArgumentsImpl (com.jetbrains.lang.dart.psi.impl)
I DartArrayAccessExpression (com.jetbrains.lang.dart.psi)
C DartArrayAccessExpressionImpl (com.jetbrains.lang.dart.psi.impl)
I DartAsExpression (com.jetbrains.lang.dart.psi)
C DartAsExpressionImpl (com.jetbrains.lang.dart.psi.impl)
I DartAssertStatement (com.jetbrains.lang.dart.psi)
C DartAssertStatementImpl (com.jetbrains.lang.dart.psi.impl)
I DartAssignExpression (com.jetbrains.lang.dart.psi)
C DartAssignExpressionImpl (com.jetbrains.lang.dart.psi.impl)
I DartAssignmentOperator (com.jetbrains.lang.dart.psi)
C DartAssignmentOperatorImpl (com.jetbrains.lang.dart.psi.impl)
I DartAwaitExpression (com.jetbrains.lang.dart.psi)
C DartAwaitExpressionImpl (com.jetbrains.lang.dart.psi.impl)
I DartBitwiseExpression (com.jetbrains.lang.dart.psi)
C DartBitwiseExpressionImpl (com.jetbrains.lang.dart.psi.impl)
I DartBitwiseOperator (com.jetbrains.lang.dart.psi)
C DartBitwiseOperatorImpl (com.jetbrains.lang.dart.psi.impl)
I DartBlock (com.jetbrains.lang.dart.psi)
C DartBlockImpl (com.jetbrains.lang.dart.psi.impl)
I DartBreakStatement (com.jetbrains.lang.dart.psi)
```

Figure 2 - 3: Generated List of PSIElement Implementation and Extendation Across The Platform.

[8] Analysis Phase: it the start phase of compilation, it has a lexer and a parser.

kind of expression in every language implements it. The image is generated using the **Find tool of intellij Idea**.

2.3.2 Editor Functionalities

The editor of intellij idea (**com.intellij.openapi.editor**) is not a simple editor, it has multiple well-known features which will not be discussed, what is important is that it uses events and actions to trigger it's features. The editor has multiple fields each one of them represents an element of the editor, like the document[9], panels, caret model[10], inlay model[11], folding model. Each one uses PSI elements to communicate with each other.

2.3.3 Events

Intellij idea is an event driver platform, it uses events for features associated with the editor, syntax highlighting, coloring, hints, documentation, auto-completion etc…, events are associated with **actions**, for example, a syntax error occurred (Event) will trigger the red underline (Action) which in its turn uses PSI to apply what should be done [11].

2.3.4 Actions

Actions are the major user of PSI, every button and every event is implemented as an action. Actions represents the actual code any feature or button in the platform. Actions have a specific system that works to pin the button to specific location in the platform, implement the listeners, and pin them to the button and specify the name of the button and everything related to buttons or actions. The action system is very flexible to the point that the possibilities and capabilities are countless.

2.3.4.1 Action System

Every action implements an abstract class called AnAction in the package openapi (com.intellij.openapi.actionSystem.AnAction). For example, "one of the action classes is responsible for the **File | Open File**... menu item and the Open File toolbar button. The action implementation determines the contexts in which an action is available, and its functionality when selected in the UI. Registration determines where an action appears in the IDE UI. Once implemented and registered, an action receives callbacks from the IntelliJ Platform in response to user gestures".[12]

2.4 Dart

Dart source code is a main actor in this project, the Dart source code PSI structure was used to implement the Feature, few packages were required from dart source code. the packages are: (**com.jetbrains.lang.dart.analyzer**) and (**com.jetbrains.lang.dart.DartLanguage**) and (**com.jetbrains.lang.dart.psi.***), the analyzer is used to recognize the static type of

[9] Document: it represents the text of the code as a string and is used to manipulate the code as a string.
[10] Caret model: caret is the pointer used in the editor.
[11] Inlay model: inlay model represents the inlay texts in general, with all of its properties.

any variable or parameter, which makes it the most important class in the project. (DartLanguage) is used to recognize the file type.

2.5 Conclusion

This section must have qualified the reader to at least understand the diagrams in the following section. This section taught the reader how intellij idea components work together. as said, intellij Idea is a set of plugins, and every plugin is a set of one or more actions and events, and every triggers an action, and every action is implemented in association with PSI layer of the language if the Plugin is language specific or the PSI layer of the platform if the plugin represents a general functionality in the IDE. This section also discussed the importance of PSI.

Section 3: Analysis And Design of The System

3.1 Introduction

This section will discuss the plugin as a system and its analysis, there will be diagrams and explanations for the diagrams. Also this section will take a deep dive through the plugin structure and how Dart libraries communicate with intellij idea.

3.2 System Analysis

System Analysis is a phase of development in which we study the parts of a system theoretically and its problems and discuss ways to improve the system.

3.2.1 Diagram

3.2.1.1 Use Case Diagram

A use case diagram is a behavior diagram in UML. Use case diagrams model the functionality of a system using actors and use cases. [14]

Use case diagram contains multiple elements, each used one will be defined:

- **Actor**

  It is an element represents any thing interact with the system, could be a user, or a sub system. In our case there is a primary actor which is the user, and multiple supporting actors. They are located out side the system in the right or left.

- **System**

  It is the element that contains all elements interactions with each other through use cases and relationships.

- **Use Case**

  Use cases are requirements, primarily functional or behavioral requirements that indicate what the system will do. They are represented by an oval shape.

- **Extend Relationship**

  Extend relationship point to an optional relationship, it indicates that use case A could trigger use case B (Not A Must). It is represented by a pointed arrow from use case B (Extended) to use case A (Primary).

- **Include Relationship**

Include Relationship point to a mandatory relationship between use case A and use case B. it indicates that if use case A is triggered. it must trigger use case B implicitly. It is represented by a pointed dotted arrow from use case A to implicit use case B.

The diagram in [Figure 3 – 1] represents the use case diagram of the whole system. It contains 6 actors which will be discussed in details

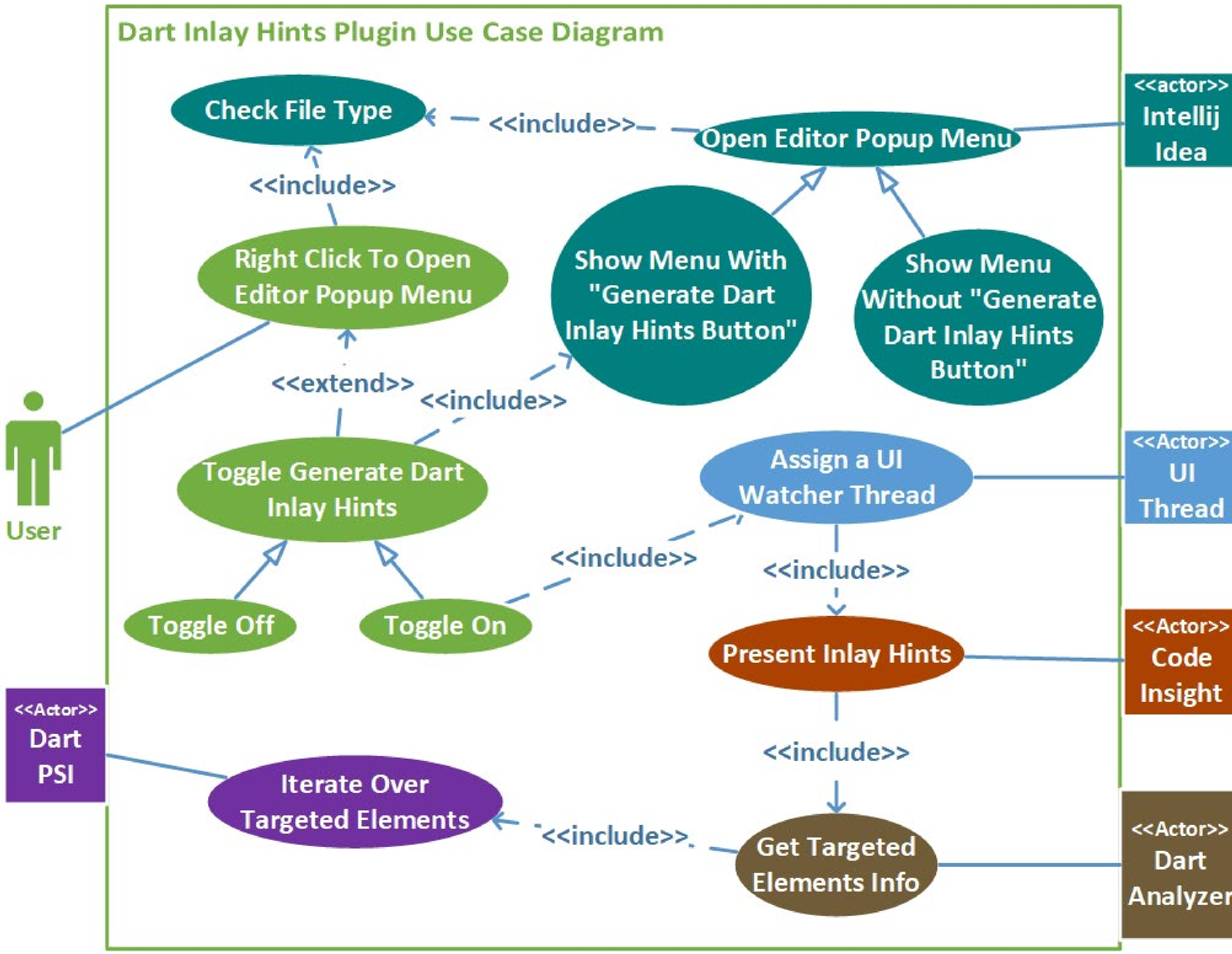

Dart Inlay Hints Plugin Use Case Diagram
Check File Type
<<include>>
Open Editor Popup Menu
<<actor>> Intellij Idea
<<include>>
Right Click To Open Editor Popup Menu
Show Menu With "Generate Dart Inlay Hints Button"
Show Menu Without "Generate Dart Inlay Hints Button"
<<extend>>
<<include>>
User
Toggle Generate Dart Inlay Hints
Assign a UI Watcher Thread
<<Actor>> UI Thread
<<include>>
<<include>>
Toggle Off
Toggle On
Present Inlay Hints
<<Actor>> Code Insight
<<Actor>> Dart PSI
<<include>>
Iterate Over Targeted Elements
<<include>>
Get Targeted Elements Info
<<Actor>> Dart Analyzer

Figure 3 - 1: System Use Case Diagram

- **User**

As the diagram in [Figure 3 – 2] represents, the user (The Main Actor) by itself has two functionalities in this system. the first is to right click inside the editor and the second is to toggle the hint generation action through a click, which is optional; the user can simply just open the menu. If the user right-click to open the menu, then another actor will come in contact which is (intellij Idea).

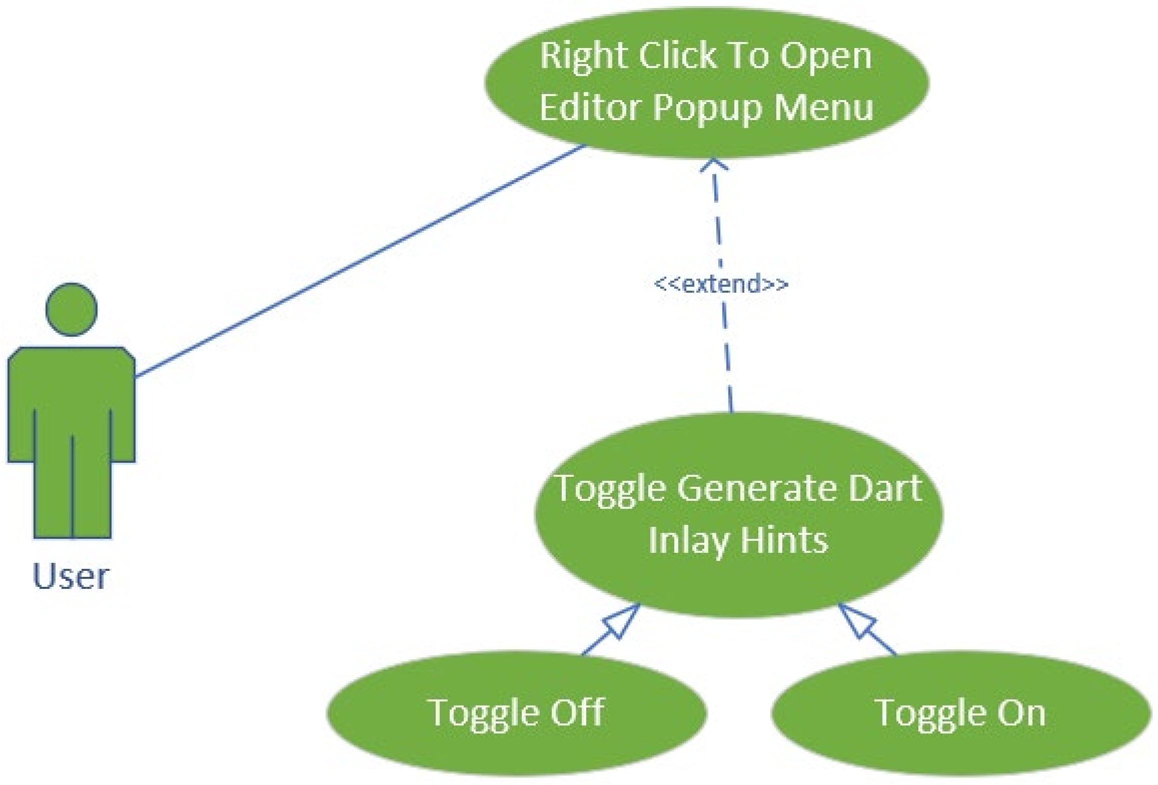
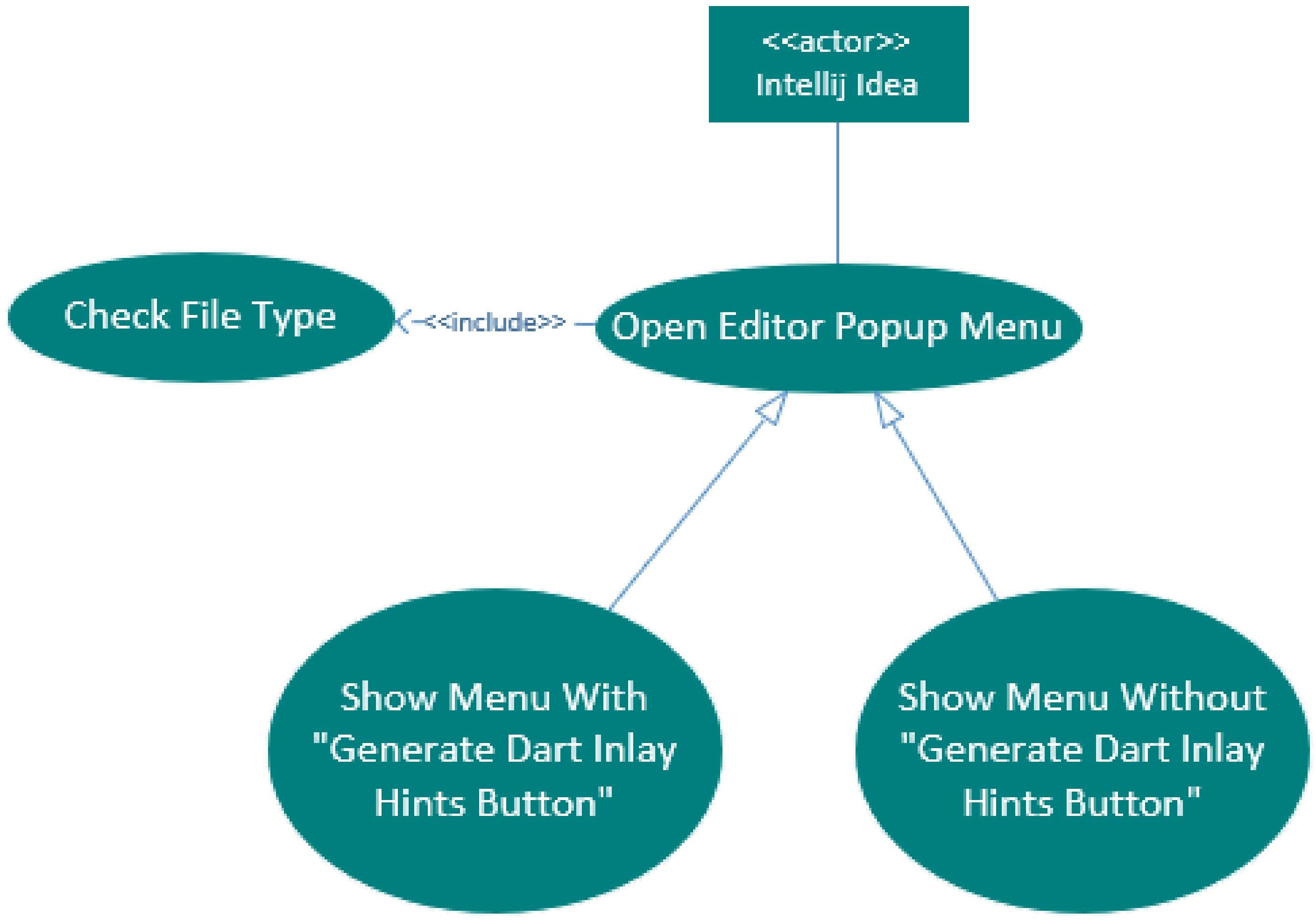

Right Click To Open Editor Popup Menu
<<extend>>
User
Toggle Generate Dart Inlay Hints
Toggle Off
Toggle On

Figure 3 - 2: User Use Case Diagram

- **Intellij Idea**

Intellij Idea as an actor has 1 single functionality, it determines whether the **EditorPopupMenu** should have the (Generate Dart Inlay Hints) button or not. It does show the button when the current opened file is a .dart file. When the current file is not a .dart file the button is not shown. This functionality is done through checking the type of the file as PSIFile Element. If the PSIElement it is an instance of a Dart Language File then show the button. If not then don’t.



<<actor>>
Intellij Idea
Check File Type
<<include>>
Open Editor Popup Menu
Show Menu With "Generate Dart Inlay Hints Button"
Show Menu Without "Generate Dart Inlay Hints Button"

Figure 3 - 3: Intellij Idea Use Cases

- **Dart PSI**

Although the Dart PSI actor has a single use case, it does a crucial action in terms of functionality. It iterates over the code as PSI elements using the function **DartRecursiveVisitor()**. It reads the code element by element to and send the element to Dart Analyzer which does the magic of extracting the Static types of the Variables, and methods parameters.

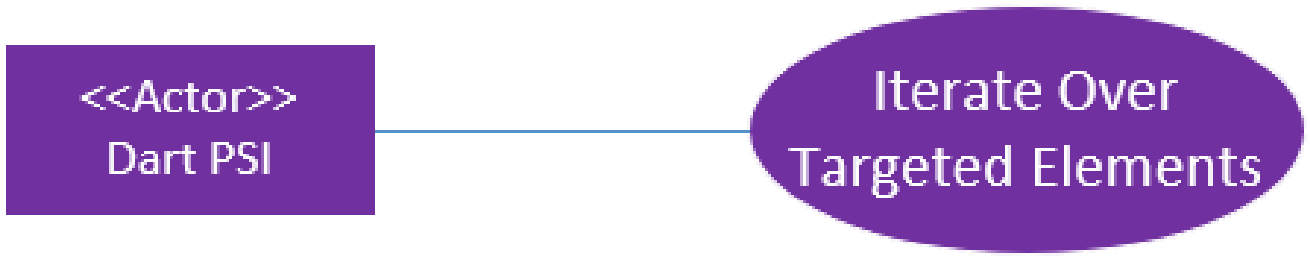


Figure 3 - 4: Dart PSI Use Case

- **Dart Analyzer**

The dart Analyzer takes the PSI elements one by one and extract information from it. information such as static type, element offset related to the document, element file path, and other info. This information later will be sent to CodeInsight for hints providing.

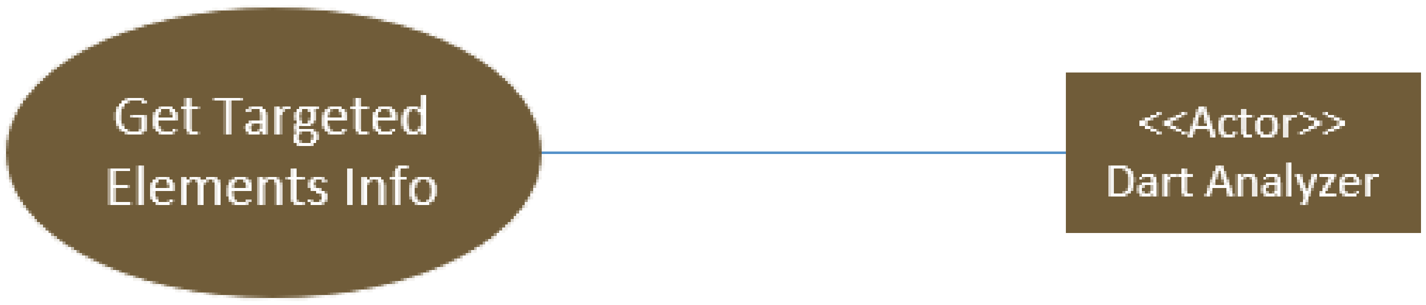


Figure 3 - 5: Dart Analyzer Use Case

- **Code Insight**

The [Figure 3 – 6] represents the actor that adds the inlay hints to the editor, it works by retrieving the texts it should add to the editor from the dart analyzer as Strings, it wrap the text inside an element that is represented in the editor as a rendered visual element called (**Inlay**). The wrapper class which wrap the text as inlay is called (**PresentationRenderer**).

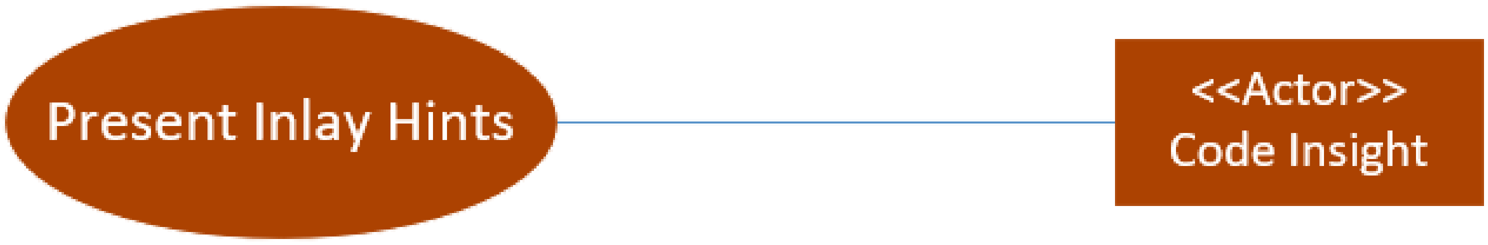


Figure 3 - 6: Code Insight Use Case

- **UI Thread**

When the user Clicks the button that activates the dart hints, a UI Thread start running, the thread always iterates over the code to check whether a change occurred or not. If a change occurred, the PSI element is sent over to the analyzer and then to the **codeInsight** which will present the new hints to the element in the editor.

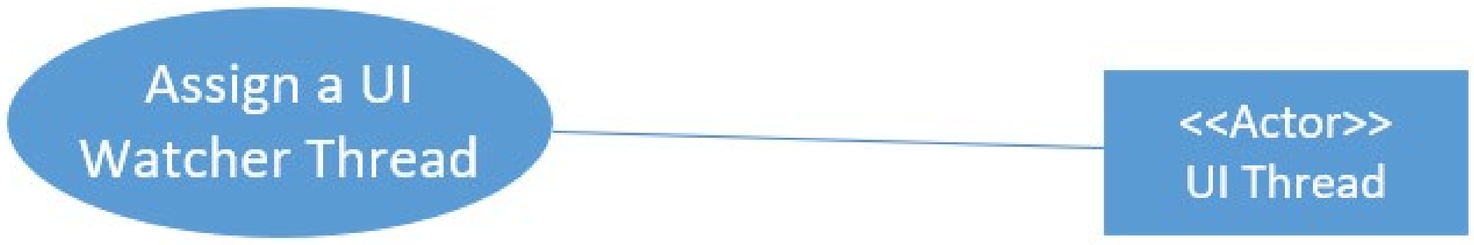


Figure 3 - 7: UI Watcher Thread

3.2.1.2 Functional Flow Block Diagram

Functional Flow Block Diagram (FFBD) is a diagram used to show the sequential relationship of all functions that must be accomplished by a system. The FFBD is functionally oriented. The process of defining lower-level functions and sequencing relationships is often referred to as functional decomposition. It allows traceability vertically through the levels [14].

FFBD allows you to design a diagram in levels, level 1 starts from the start of system, level 2→ level n-1 represents how system code is fragmented, as each level represents n number of functionalities happens sequentially to each other.

Functional Flow Block Diagram elements are many, and few has been used [14]:

- **Function Block**

  Function Block indicates a functionality in the system and not necessarily an actual function in the code. and it is represented by a square with the functionality in it.

Generate Dart Hints

Figure 3 - 8: Functional Block Representation

- **Flow Arrow**

  The flow refers to the sequence of functions and their logical position in the system, it is represented by a pointing arrow from A to B.

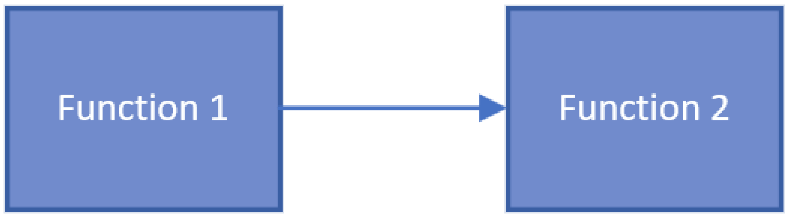


Figure 3 - 9: Flow Representation

- **Decomposition**

  Decomposition indicates a sub sequence of function in a deeper level (Scope) of code, it is represented by a dotted line from the primary function to the first in the sub sequence and last in the sub sequence.

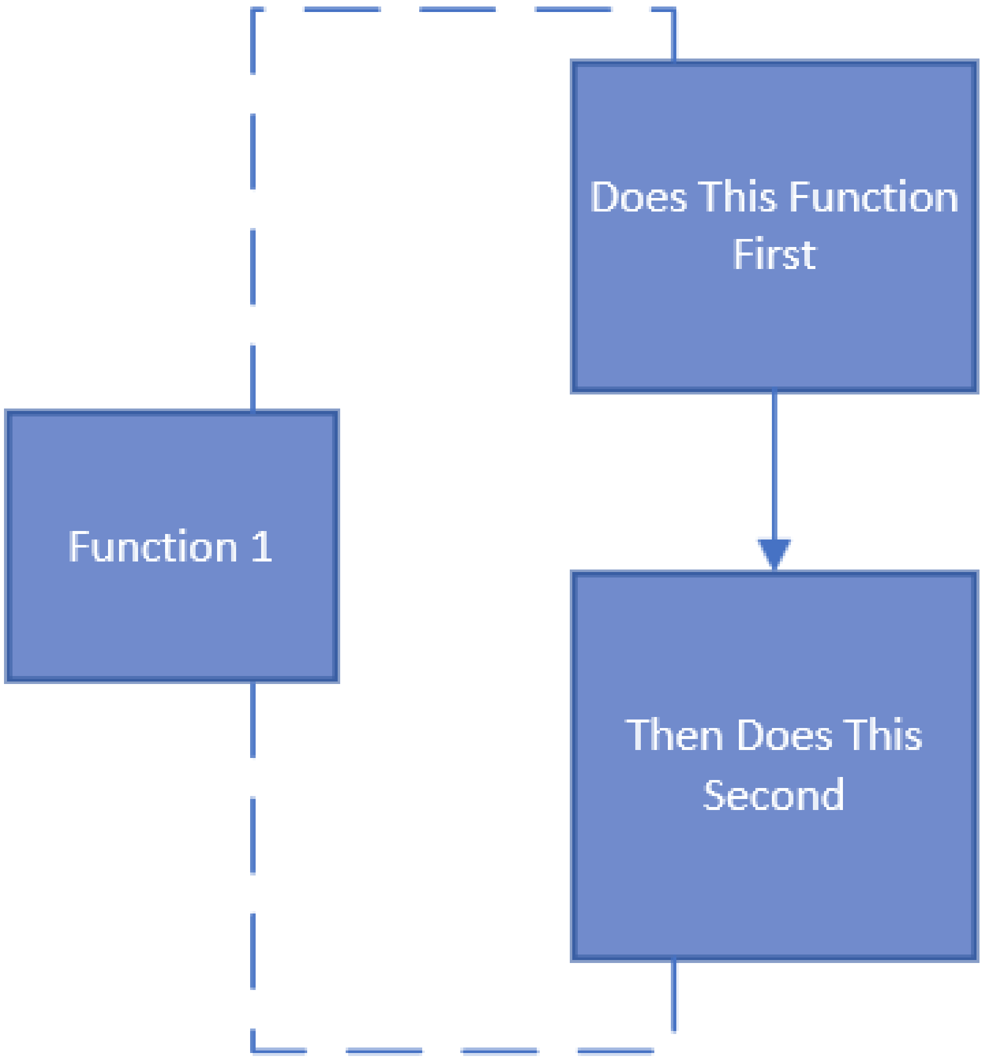


Figure 3 - 10: Decomposition Representation

- **Parallel Block**

Parallel Block indicates that 2 functions will happens in parallel, and it is represented by n functions with an arrow coming to each function from an AND symbol and also arrow coming out of functions to AND symbol to represents the end.

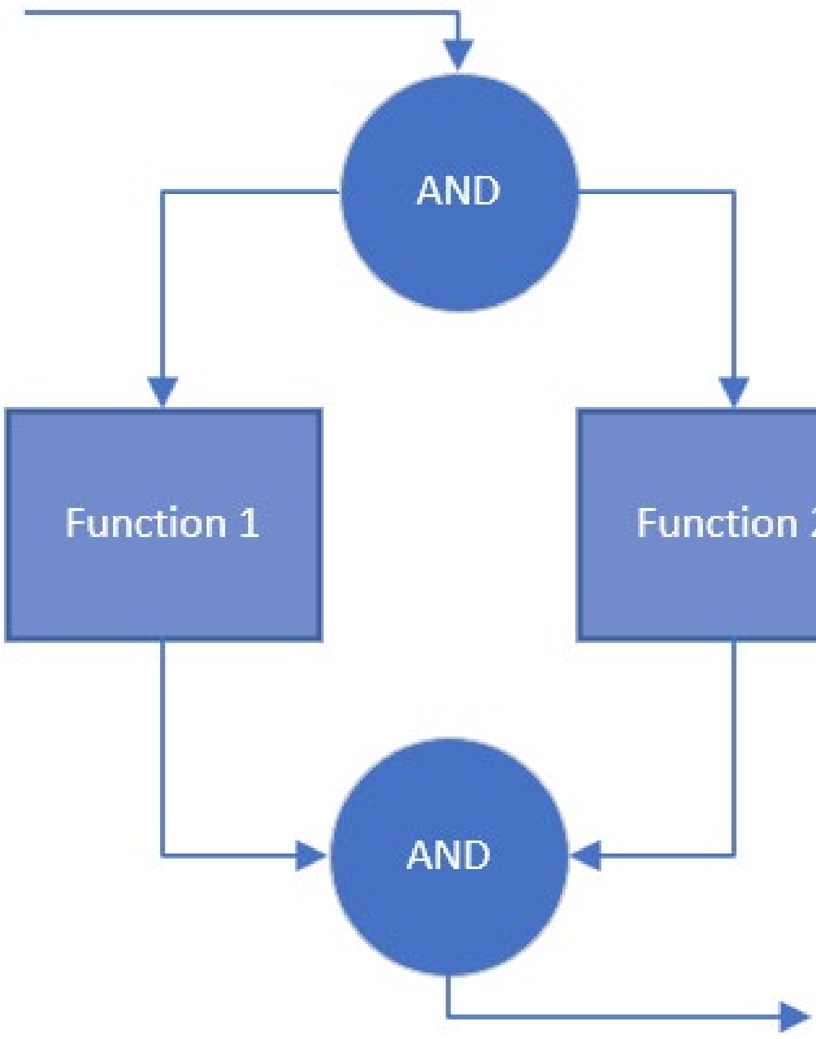


Figure 3 - 11: Parallel Block Representation

- **Iteration**

Iteration element is an element that represents a pipe line of n functions which must be invoked for unknown times.

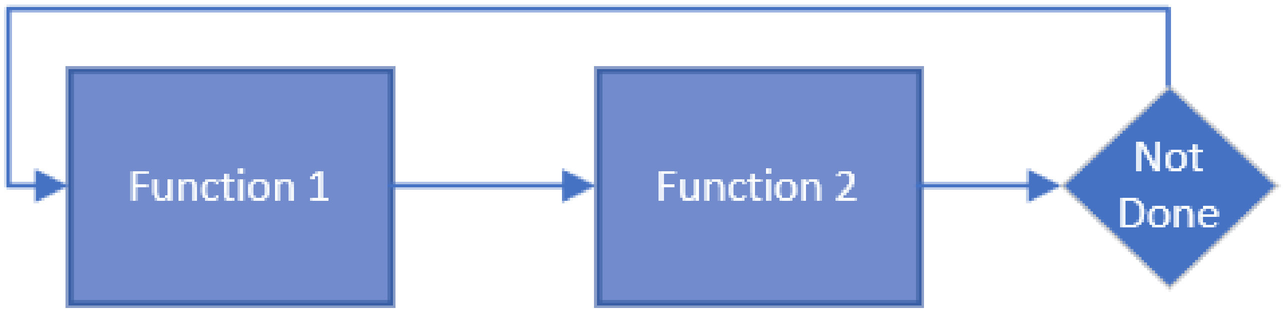


Figure 3 - 12: Iteration Representation

- **Choice**

Choice element is similar to if statement in programming, but it indicates here two functional paths one if them must be taken, Go and Not Go indicate that if choice statement of choice is true then go to function 1 and if false then not go to function 1.

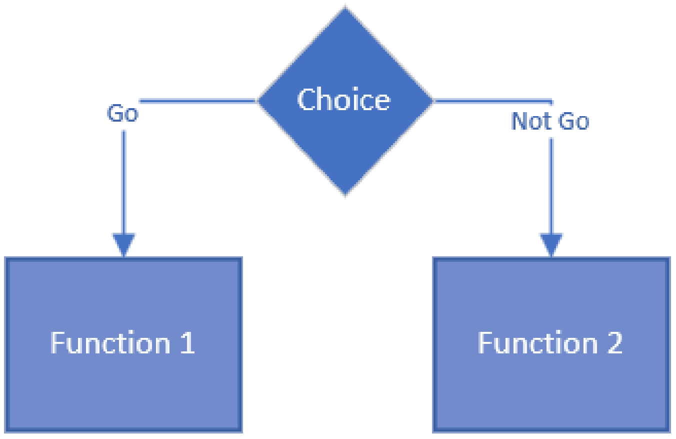


Figure 3 - 13: Choice Representation

The [Figure 3 – 14] represents the functional flow block diagram of the whole system. As the diagram says, the system is separated into 4 levels:

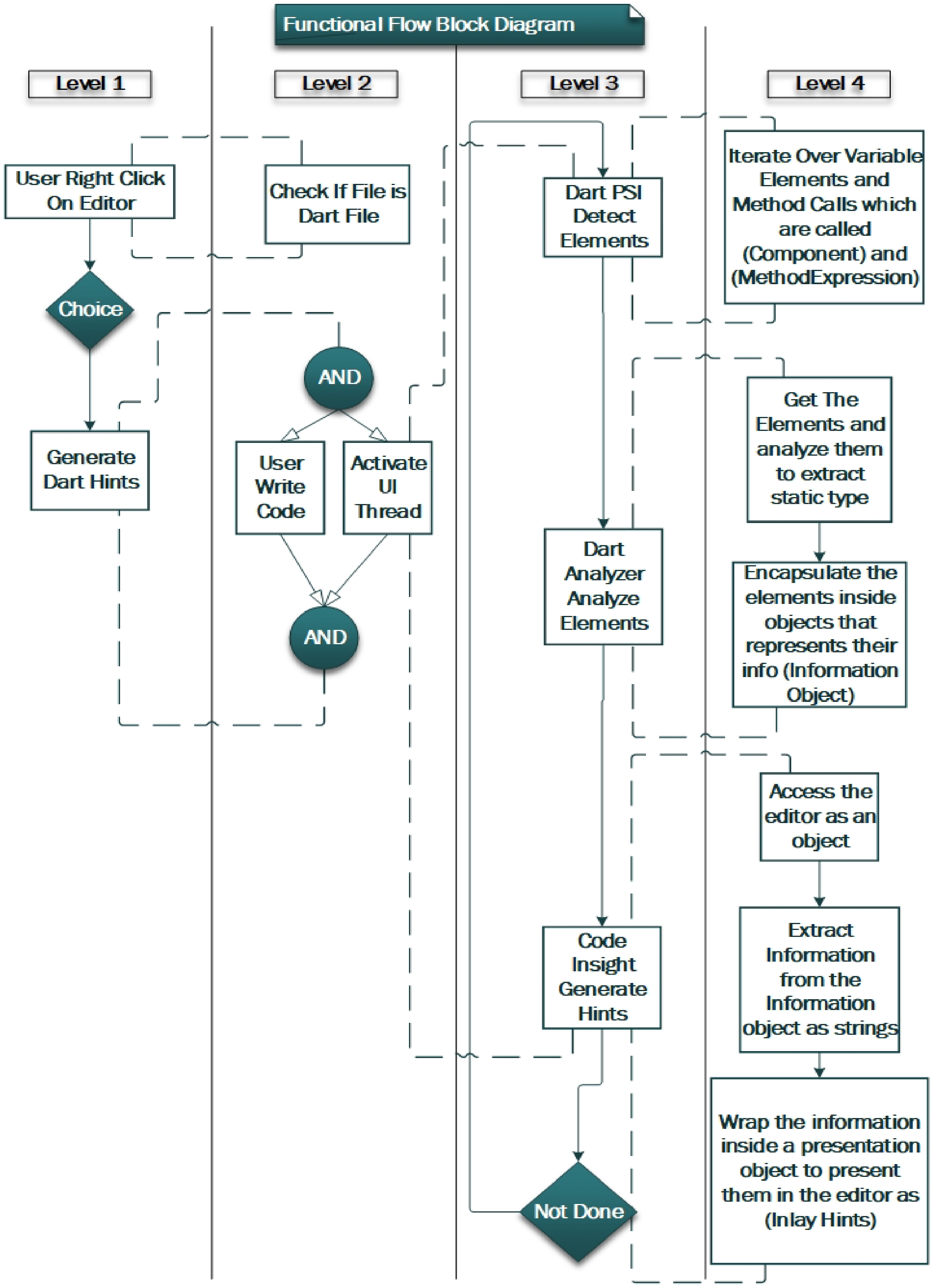

Functional Flow Block Diagram
Level 1
Level 2
Level 3
Level 4
User Right Click On Editor
Check If File is Dart File
Choice
AND
Generate Dart Hints
User Write Code
Activate UI Thread
AND
Dart PSI Detect Elements
Dart Analyzer Analyze Elements
Code Insight Generate Hints
Not Done
Iterate Over Variable Elements and Method Calls which are called (Component) and (MethodExpression)
Get The Elements and analyze them to extract static type
Encapsulate the elements inside objects that represents their info (Information Object)
Access the editor as an object
Extract Information from the Information object as strings
Wrap the information inside a presentation object to present them in the editor as (Inlay Hints)

Figure 3 - 14: System Functional Flow Block Diagram

- **Level 1**

  In the first level the user right clicks on editor to open **EditorPopupMenu** then intellij idea checks whether this is a dart file or not. If it is, then the user has a choice to click **Generate Dart Inlay Hints**. If the user does click the button, then it turns into toggled state.

- **Level 2**

  In the second level after the user clicks the button a **new thread** is created that watches the editor and its changes while the user white a code.

- **Level 3**

  In the third level as the Thread iterates over the main functions, if **Dart PSI** detect a new change in the PSI Tree of the code, the cycle of analysis starts which contains sending the statements to **Dart Analyzer**, then to **CodeInsight**.

- **Level 4**

  The forth level contains the actual functionality of the system, the **Dart PSI** iterates over the code to check if there are any changes in code, if any, then sends the statements that the inlay hints should be added to- to **Dart Analyzer** for analysis, then The function of **CodeInsight** will handle the information is sent to it and wrap it as inlay hint for presentation in the editor.

3.2.1.3 Sequence Diagram

Sequence Diagram is an interaction diagram that models a single scenario executing in a system it is usually related to a requirement use case.

The Sequence Diagram in [Figure 3 – 15] represents how inlay hints are added to the editor. At first, the UI thread ask the PSI to look for a Change in PSI Tree. Then an alternative occurs, if there are no change in the previous PSI scan then no need to proceed, if any change occurred, then send elements for analysis then to code insight for wrapping information as inlay hints, then return result to user inside the editor.

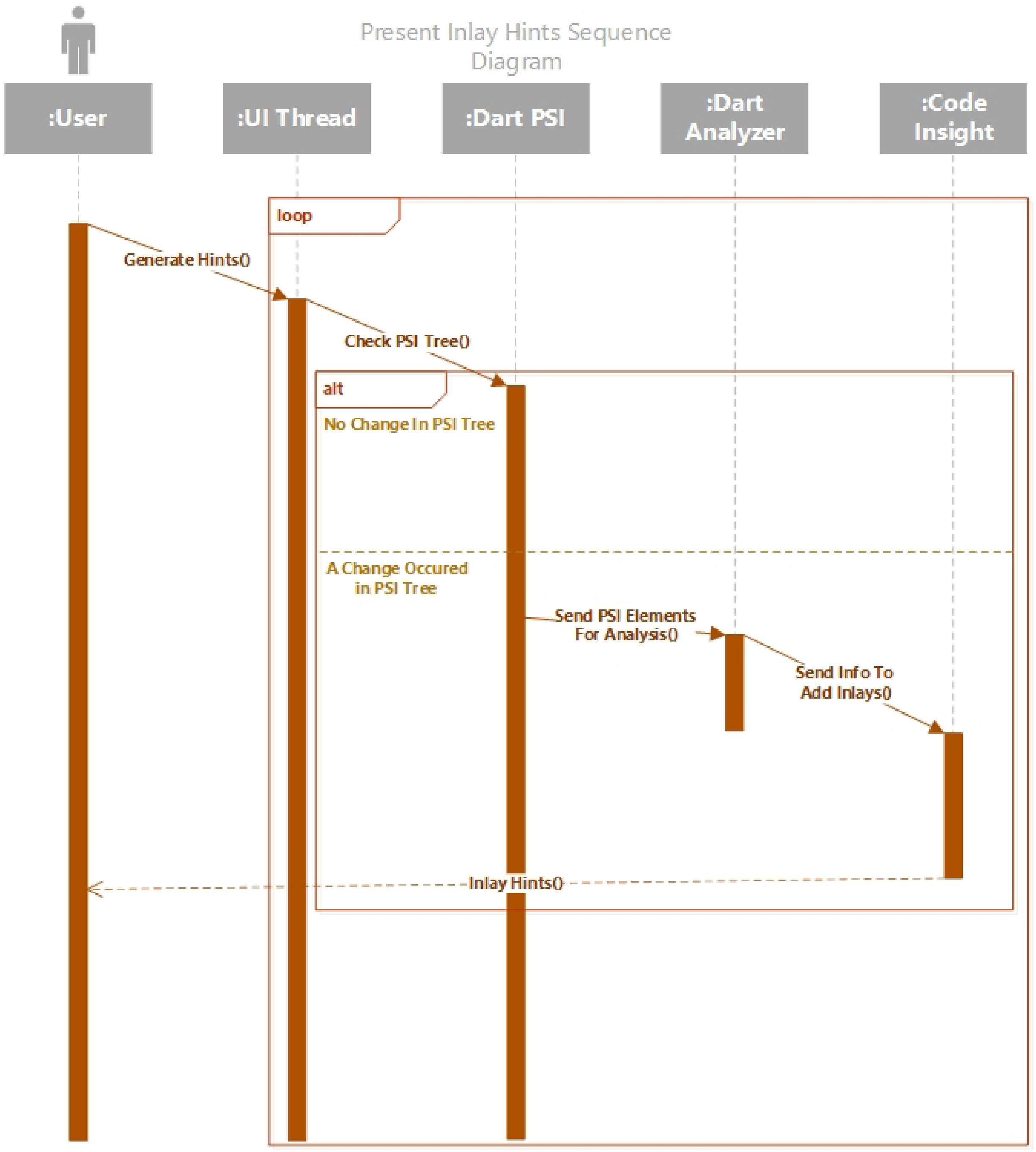


Figure 3 - 15: Present Inlay Hints Sequence Diagram

3.3 Conclusion

This chapter have discussed the analysis and design of the system. as said in use case diagram, there are six actors. user, intellij idea, UI Thread, Dart PSI, Dart Analyzer, Code Insight. This section discussed the role of each actor in the system. Functional Flow Block Diagram, have discussed how functions of the system interact with each other in levels. In sequence diagram, it has discussed how interactions between actors for present inlay hints use case.

Section 4: Used Languages and Implementation Analysis

4.1 Introduction

This section will dive deep into how the plugin was implemented and what languages used to accomplish the task.

4.2 Used Languages

- **XML**

  The Extensible Markup Language (XML) is a markup language and file format for storing, transmitting, and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is for representing structured information such as: documents, data, configuration, books, transactions, invoices, and much more. [15] In dart inlay hints project it has been used to specify some properties of the plugin during development.

- **Java & Kotlin**

  java is used to code the actual functionality of the plugin alongside kotlin because intellij idea platform plugins are written in java and kotlin. Kotlin is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library. Kotlin is used in dart inlay hints pluglin for two tasks, first is configure a plugin file that is written in kotlin, second is using kotlin classes and interface, and reading kotlin-written platform source code.

4.3 Plugin Structure Analysis

JetBrains have a complete system for building plugins, this sub-section will discuss how dart inlay hints used the system to accomplish the task of the plugin.

To start, intellij idea does have a plugin template in create project section. plugins are written using only two lanuages, java and kotlin, and built using two tools only: Gradle, DevKit. It is recommended to use Gradle because it is more advanced and automated. Gradle role in plugin development is to execute general required tasks such as run an instance of IDE with the plugin in it for testing purposes, and wrapping the plugin code as jar file for exporting to internet purposes, etc…, Gradle build tool uses kotlin or Groovy to write its code. this project used kotlin. [16,17]

4.3.1 build.gradle.kts

a file located in the root directory of the plugin **$\build.gradle.kts**, Gradle requires a build file to be sat up, it is called **build.gradle.kts** (kts is kotlin script). This build file represents the properties of the project such as language dependencies, plugin dependencies, repositories, etc… [17], some settings that were important for this project:

plugins function add java and kotlin code support to the project as dependencies. Used to allow the plugin to be written in java or kotlin or both. **Org.jetbrains.intellij** adds intellij idea modification and building capabilities to the plugin and **org.jetbrains.changelog** adds the capability to track change in code versions later.

```
plugins {
    // Java support
    id("java")
    // Kotlin support
    id("org.jetbrains.kotlin.jvm") version "1.6.10"
    // Gradle IntelliJ Plugin
    id("org.jetbrains.intellij") version "1.4.0"
    // Gradle Changelog Plugin
    id("org.jetbrains.changelog") version "1.3.1"
}
```

Intellij function is responsible for adding dart support. adding a third party plugins into project requires a specific syntax: **PluginName:PluginVersion** just like (Dart:212.5742). This info is found under any plugin download section. The three properties at first, they represent the plugin name (**Dart Inlay Hints**) and version (**0.0.1**) and in which IDE of intellij idea the plugin will be supported (could be rider or webStorm for example, but in the case of this project it is intellij idea IDE).

```
intellij {
// Plugin properties found on Gradle.properties
    pluginName.set(properties("pluginName"))
    version.set(properties("platformVersion"))
    type.set(properties("platformType"))


    // Plugin Dependencies. Uses `platformPlugins` property from
the gradle.properties file.

plugins.set(properties("platformPlugins").split(',').map(String:
:trim).filter(String::isNotEmpty))
    plugins.add("Dart:212.5742")
}
```

4.3.2 gradle.properties

A file located in the root directory **$\gradle.properties**. the properties such as name and plugin version -which is 0.0.1 by default- are located in this file (**Gradle.properties)**. the properties file contains plugin name, IDE supported version and in this project case it is **intellij idea 2021.2.4**, platform type, and other info. [18]

4.3.3 plugin.xml

a file is located at: **$\src\main\resources\META-INF\plugin.xml**, the file plugin.xml is a file that represents the plugin level dependencies and other information related to the plugin, not to the project as a project. This twist is because the project folder could contain multiple plugins. This file is written in XML and when using Gradle, a number of metadata elements will be provided at build time by **patchPluginXml** Gradle task. This file is related to **build.gradle** in content, if a dependency needed to be used by the plugin, it must be added as a dependency to build.gradle.kts first, then to plugin.xml [19].

for this project couple of dependencies has been added, **com.intellij.modules.platform** is used to extend the platform and add a button to it. **Dart** is used to add dart source code to plugin code and access it and manipulate the dart source code as dart source code:

```
<depends>com.intellij.modules.platform</depends>
<depends>Dart</depends>
```

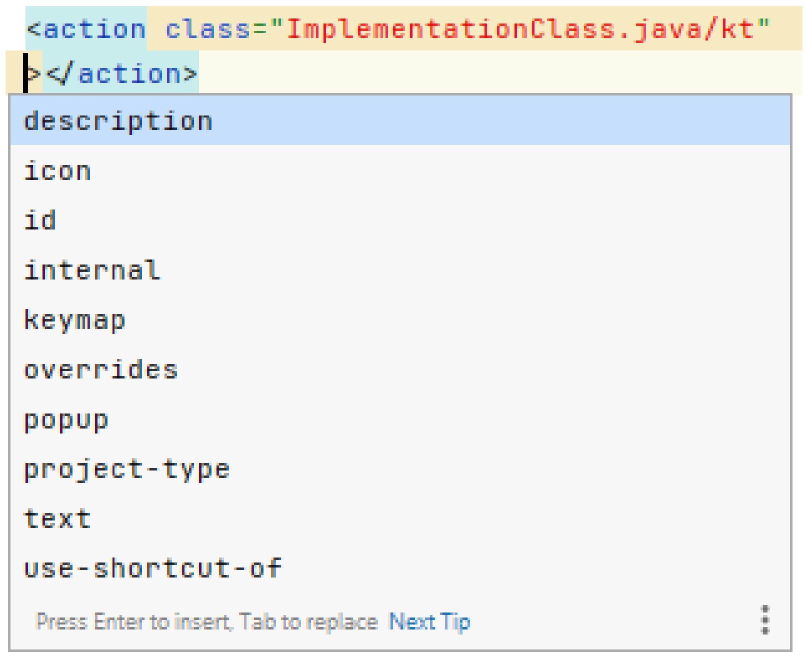
The plugin.xml file contains other info like actions, action are registered here under a tag called actions, Each action has an implementation class that must be specified and few other attributes. Implementation class represents the actual functionality of the action. [19] in the case of this project the implementation class is the class responsible for triggering the dart inlay hints:

```
<actions>
// Action assignment is here
</actions>
```

The code lines below are an example of a button being anchored to the top of EditorPopupMenu:

```
<action id="TestID" class="Path/To/Class.java\kt"
text="testButton">
    <add-to-group group-id="EditorPopupMenu" anchor="first"/>
</action>
```

the [Figure 4 – 1] represents other attributes could be assigned to an action button:



```
<action class="ImplementationClass.java/kt"
></action>
description
icon
id
internal
keymap
overrides
popup
project-type
text
use-shortcut-of
```

Figure 4 - 1: Action Properties Screenshot

4.3.4 Action Implementation

Each action has an implementation class that could be written in java or kotlin, in case of this project it has been written using java. The action implementation class inherit a class called AnAction to be validated as an action in intellij idea platform [20], and in the case of this project it, the action has implemented an interface called Toggleable in order to let the action has an On/Off state:

```
public class DartInlayHints extends AnAction implements
Toggleable {

// Action (Button/Trigger) Code is Here

}
```

The **AnAction** class has multiple abstract methods that could be overridden, but there are two major methods that must be overridden by every IntelliJ Platform action. They are **AnAction.update()** and **AnAction.actionPerformed()**.

- **AnAction.update()**

  AnAction's method **AnAction.update()** is called by the IntelliJ Platform framework to update an action state. Action state means the state of visibility and enability (enabled, visible) in the UI of an IDE and other states such as icon. An object of the **AnActionEvent** type is passed to this method and contains information about the current context for the action which has been used in order to achieve two tasks, the first is a button that get hidden when the button is not a **.dart** file, and the second task is changing toggle icon between toggled and not toggled. So, Actions are made available by changing state in the Presentation object associated with the event context. As explained in Overriding the **AnAction.update()** Method, it is vital **update()** methods execute quickly and return execution to the IntelliJ Platform [20].

  The **AnActionEvent** object passed to **update()** carries information about the current context for the action. Context information is available from the methods of **AnActionEvent**, providing information such as the Presentation and whether the action is triggered. Additional context information is available using the method **AnActionEvent.getData()**. Keys defined in **CommonDataKeys** [12]are passed to the **getData()** method to retrieve objects such as Project, Editor, PsiFile, and other

[12] CommonDataKeys: it is a class that contains attributes to extract information from the current context of the event. Such as execution file, current editor, etc…

information. Accessing this information is relatively light-weight and is suited for **AnAction.update()**.

the statements below are responsible for the enabling and disabling depending on the file type and changing it's toggle state.

```
@Override
public void update(@NotNull AnActionEvent e) {
    PsiFile psiFile = e.getData(CommonDataKeys.PSI_FILE);
    assert psiFile != null;

e.getPresentation().setEnabledAndVisible(psiFile.getLanguage().i
s(DartLanguage.INSTANCE));
    Toggleable.setSelected(e.getPresentation(), State);
}
```

- **AnAction.ActionPerformed()**

An action's method **AnAction.actionPerformed()** is called by the IntelliJ Platform if available and selected by the user. This method does the heavy lifting for the action - it contains the code executed when the action gets invoked. The **actionPerformed()** method also receives **AnActionEvent** as a parameter, which is used to access projects, files, selection, etc… When the user selects an enabled action, be it from a menu or toolbar, the action's **AnAction.actionPerformed()** method is called. This method contains the code executed to perform the action, and it is here that the real work gets done [20].

By using the **AnActionEvent** methods and **CommonDataKeys**, objects such as the Project, Editor, PsiFile, and other information is available. For example, the **actionPerformed()** method can modify, remove, or add PSI elements to a file open in the editor.

```
@Override
public void actionPerformed(@NotNull AnActionEvent e) {
// Action (Generating Dart Inlay Hints) code is here.

}
```

There are other methods to override in the AnAction class, such as changing the default Presentation object for the action. There is also a use case for overriding action constructors when registering them with dynamic action groups, demonstrated in the

Grouping Actions tutorial. However, the update() and actionPerformed() methods are essential to basic operation.[20]

dart inlay hints **AnAction.ActionPerformed()** method contains couple of **CommonDataKeys**. The project relies on the statements below to access the elements of current file and editor in which hints are supposed to be added to:

```
PsiFile psiFile = e.getData(CommonDataKeys.PSI_FILE);
Editor editor = e.getData(CommonDataKeys.EDITOR);
```

psiFile.accept() is used to iterate over current file elements in general, it's parameter is a visitor of a specific language. This project managed to iterate over dart elements. This is done through an interface being implemented anonymously from **DartPSI** to iterate over the Dart code elements:

```
psiFile.accept(new DartRecursiveVisitor() {
// iteration over elements is done here recursively.
}
```

for this project, there were two visitors, one for variable declaration and a second for methods call expressions. So, the elements the visitor must iterate over are **DartComponent** and it represents variables, **DartCallExpression** represents method calls. The visitors iterates over them and start the cycle of presenting hints to editor as discussed in section 3:

```
@Override
public void visitComponent(@NotNull DartComponent o) {
// iteration over variables is here
}

@Override
public void visitCallExpression(@NotNull DartCallExpression o) {
// iteration over call expressions is here
}
```

When a visitor finds a change in an element it has visited, the visitor sends the element to analyzer to get types using following statements:

```
List<HoverInformation> hoverList = psiFileVirtualFile != null ?
DartAnalysisServerService.getInstance(psiFile.getProject()).anal
```

```
ysis_getHover(psiFileVirtualFile, o.getTextOffset()) :
Collections.emptyList();

var information = hoverList.isEmpty() ? null :
(HoverInformation)hoverList.get(0);
```

then targeted information get extracted through **information.getStaticType();** and get wrapped inside a specific type of object in order to make it acceptable for the editor. The statement below represents the wrapping of information inside a presentation object:

```
var presentation = new
PresentationFactory(editor1).smallText(text);
```

Next, editor will get the presentation and add it to the editor using **editor.getInlayMode().addInlineElement()**. the editor is accessed and an inline element has been added through getting the offset of the element in the editor and wrap the presentation created earlier inside a rectangle for presentation purposes and then add it to the editor through the inlay model:

```
editor.getInlayModel().addInlineElement(offset, false, new
PresentationRenderer(
        new
PresentationFactory(editor1).roundWithBackground(presentation)
));
```

This cycle is repeated through creating a UI thread that checks the elements for changes every few seconds, when a change has occurred the cycle starts over. This trick is done using the following statement:

```
EdtExecutorService.getScheduledExecutorInstance().scheduleWithFi
xedDelay(Task, 1, 5, TimeUnit.SECONDS);
```

4.4 Conclusion

This section has discussed multiple things related to actual implementation of the project. Such as plugin structure, and how events and actions are truly implemented which is by inheriting a class named **AnAction**, and overriding its two major functions, **actionPerformed()**, **update()**. This section has also discussed how editor and files are accessed which is using **CommonDataKeys**. To wrap up how the action has been implemented, firstly an iteration through variables and call expression elements is done using **DartRecursiveVisitor()** class, it has visited two elements: **DartComponent** which represent the variables, and **DartCallExpression** which represents the call expression in the code. When an element is found and it needs inlay hints to be added -need here means the element does not have an inlay hint or it has changed-, it sends the element to the analyzer and get information returned, after that the **Static Type** get extracted and wrapped inside a **presentation** object to make it suitable for inlay hint purpose, then the presentation is wrapped again inside a rectangle, then the whole object is added to the editor by accessing the editor object and its inlay model using **editor.getInlayModel()**.

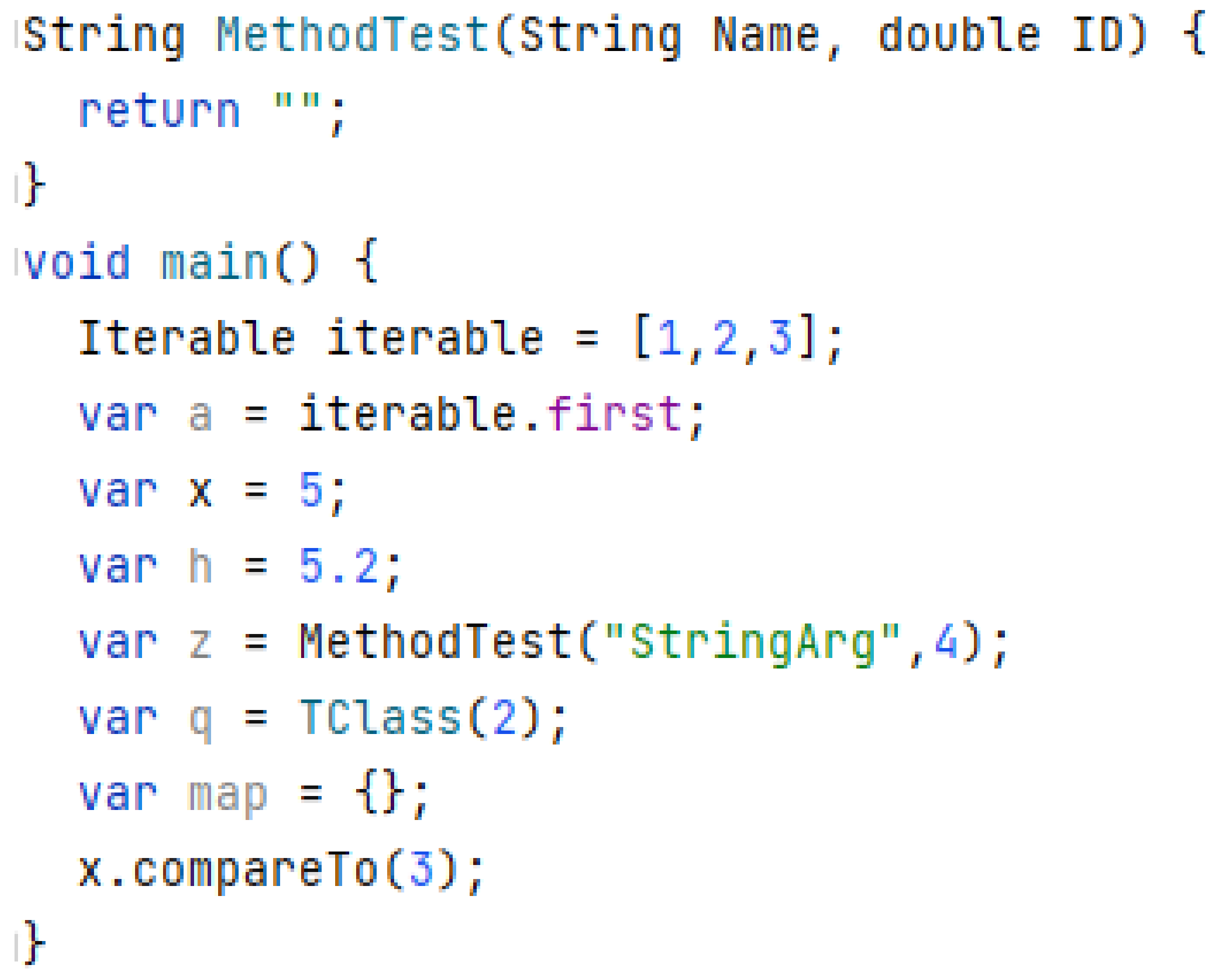
Section 5: Implementation Results and Discussion

5.1 Introduction

This section will discuss plugin implementation results.

5.2 Results

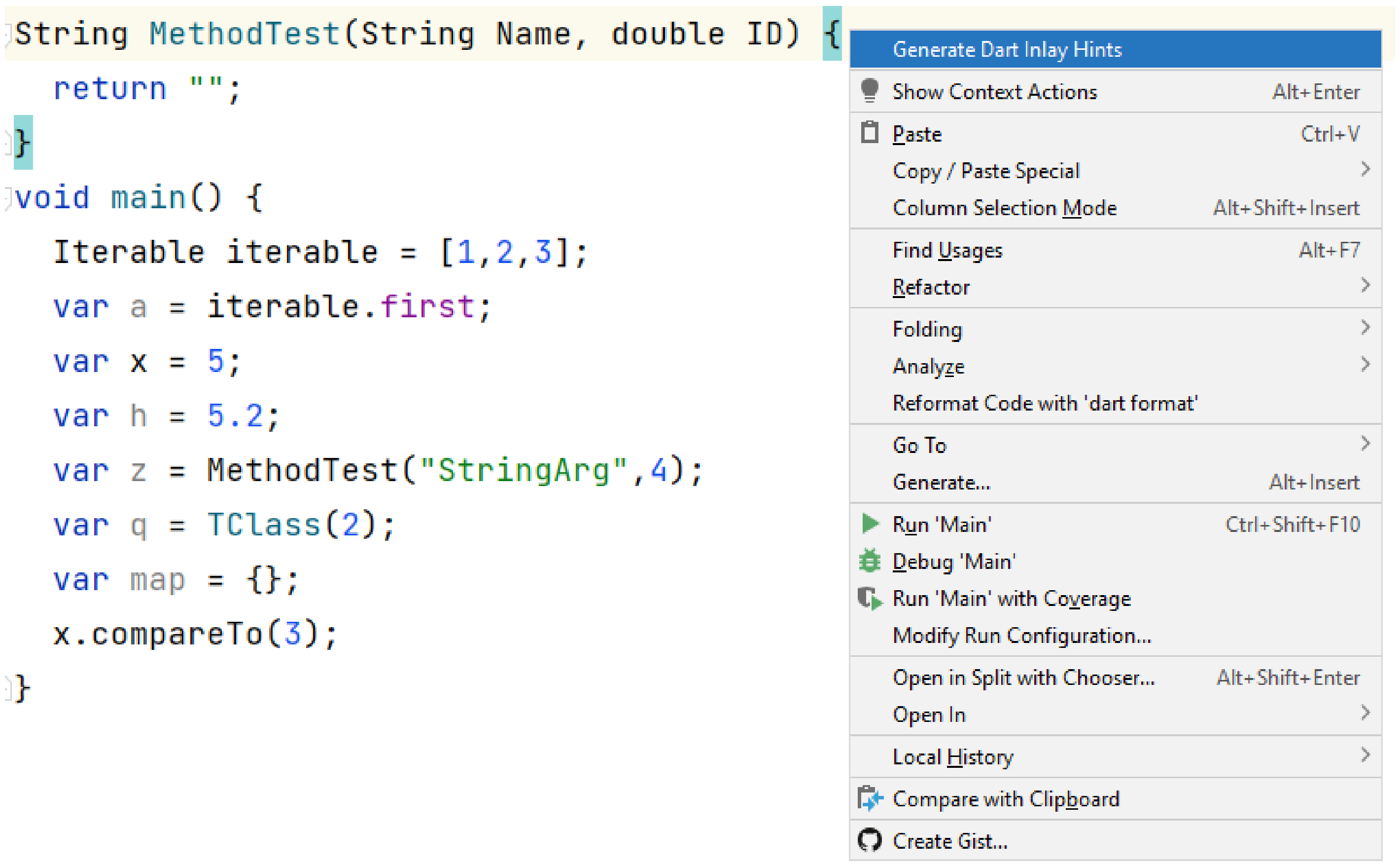
At first when the IDE starts, the user needs to have an editor opened on Dart file:



```dart
String MethodTest(String Name, double ID) {
  return "";
}
void main() {
  Iterable iterable = [1,2,3];
  var a = iterable.first;
  var x = 5;
  var h = 5.2;
  var z = MethodTest("StringArg",4);
  var q = TClass(2);
  var map = {};
  x.compareTo(3);
}
```

Figure 5 - 1: Opened Editor on Dart Code

to view EditorPopupMenu the user must right click on the editor. The first button (Generate Dart Inlay Hints) is the result of this project. It is a toggleable button, if the user clicks the button, it gets toggled to on state and inlay hints are generated:



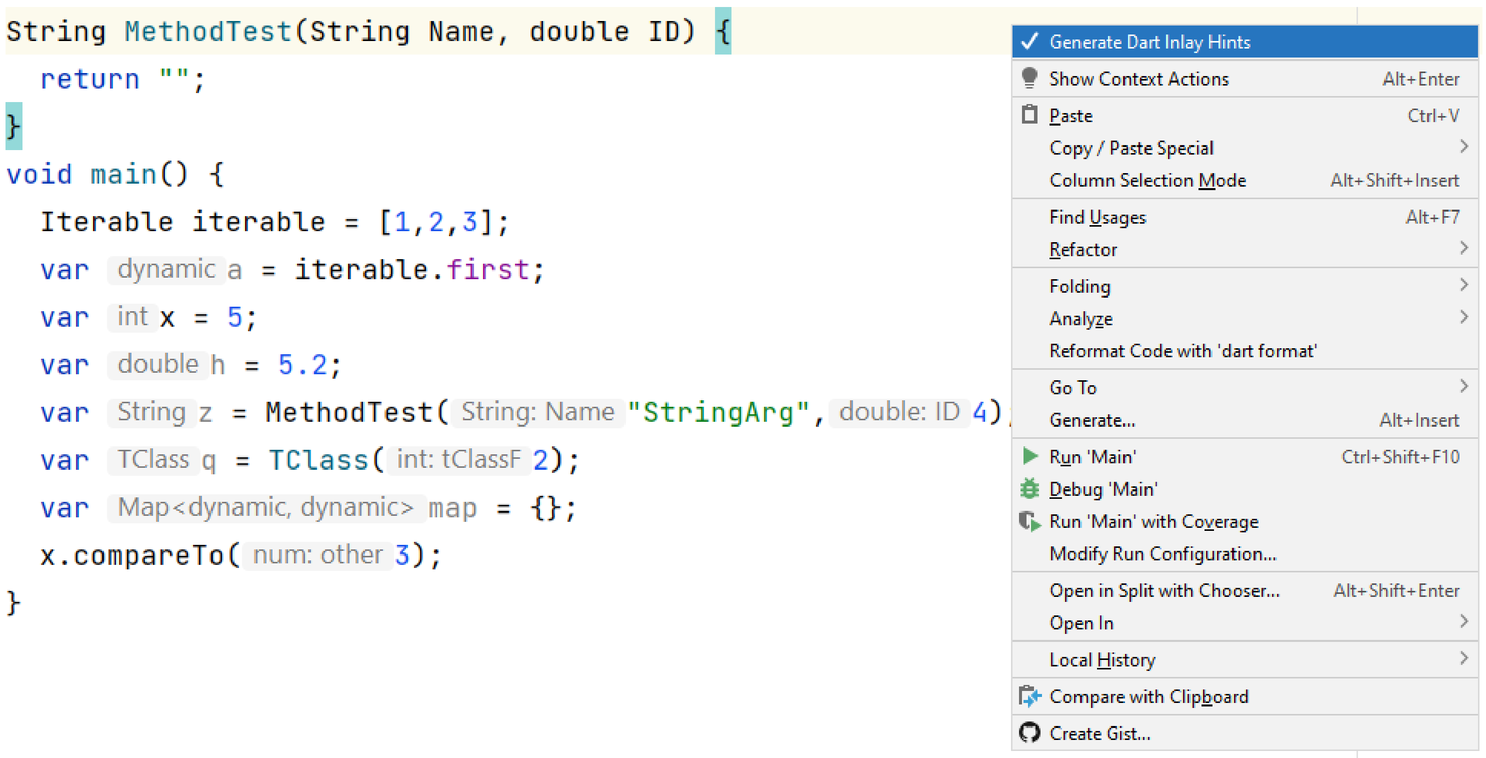
```
String MethodTest(String Name, double ID) {
  return "";
}
void main() {
  Iterable iterable = [1,2,3];
  var a = iterable.first;
  var x = 5;
  var h = 5.2;
  var z = MethodTest("StringArg",4);
  var q = TClass(2);
  var map = {};
  x.compareTo(3);
}
```

Generate Dart Inlay Hints
Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Analyze
Reformat Code with 'dart format'
Go To
Generate... Alt+Insert
Run 'Main' Ctrl+Shift+F10
Debug 'Main'
Run 'Main' with Coverage
Modify Run Configuration...
Open in Split with Chooser... Alt+Shift+Enter
Open In
Local History
Compare with Clipboard
Create Gist...

Figure 5 - 2: EditorPopupMenu With Hints Toggle Button

As the [Figure 5 – 3] represents, there are multiple ambiguous variables, not statically typed, every variable that is not statically typed, has a type hint added to it. In the first line of the main function, there is a statically typed variable, so no hints required there.

The parameters of constructors and methods are folded into type and parameter name to add more information to the code. The plugin will detect any additional statements of course while writing code.

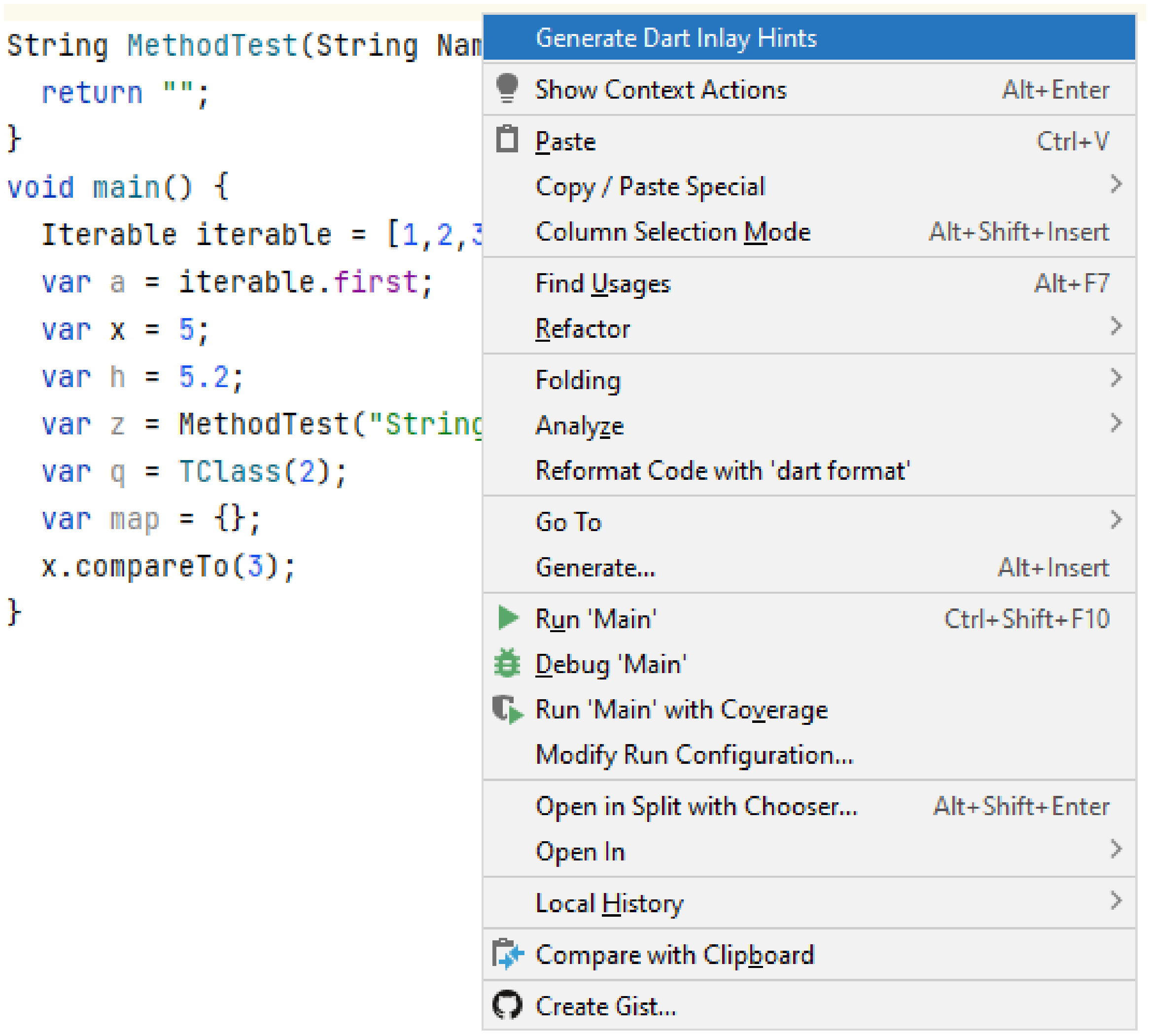


```
String MethodTest(String Name, double ID) {
  return "";
}
void main() {
  Iterable iterable = [1,2,3];
  var dynamic a = iterable.first;
  var int x = 5;
  var double h = 5.2;
  var String z = MethodTest( String: Name "StringArg", double: ID 4)
  var TClass q = TClass( int: tClassF 2);
  var Map<dynamic, dynamic> map = {};
  x.compareTo( num: other 3);
}
```

✓ Generate Dart Inlay Hints
Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Analyze
Reformat Code with 'dart format'
Go To
Generate... Alt+Insert
Run 'Main' Ctrl+Shift+F10
Debug 'Main'
Run 'Main' with Coverage
Modify Run Configuration...
Open in Split with Chooser... Alt+Shift+Enter
Open In
Local History
Compare with Clipboard
Create Gist...

Figure 5 - 3: Toggled On Hints Button

When the button is toggled again to off, the dart hints are no longer shown.

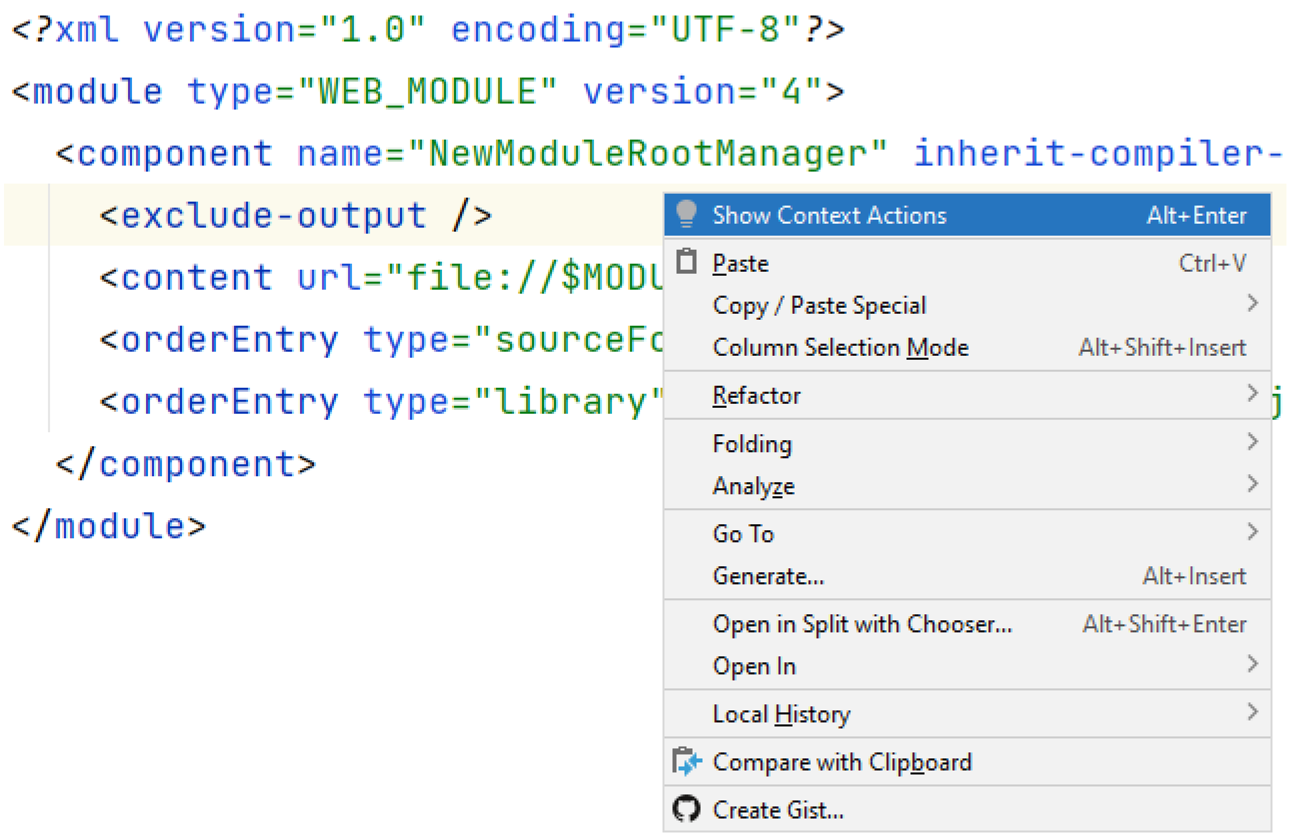


```
String MethodTest(String Nam
  return "";
}
void main() {
  Iterable iterable = [1,2,3
  var a = iterable.first;
  var x = 5;
  var h = 5.2;
  var z = MethodTest("String
  var q = TClass(2);
  var map = {};
  x.compareTo(3);
}
```

Generate Dart Inlay Hints
Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Analyze
Reformat Code with 'dart format'
Go To
Generate... Alt+Insert
Run 'Main' Ctrl+Shift+F10
Debug 'Main'
Run 'Main' with Coverage
Modify Run Configuration...
Open in Split with Chooser... Alt+Shift+Enter
Open In
Local History
Compare with Clipboard
Create Gist...

Figure 5 - 4: Toggled Off Button

This project managed to only let the button appear in a dart file. The [Figure 5 – 5] represents the EditorPopupMenu in another file. The button of generation has not appeared.



```
<?xml version="1.0" encoding="UTF-8"?>
<module type="WEB_MODULE" version="4">
  <component name="NewModuleRootManager" inherit-compiler-
    <exclude-output />
    <content url="file://$MODU
    <orderEntry type="sourceFo
    <orderEntry type="library"                              j
  </component>
</module>
```

Figure 5 - 5: Non Dart File with EditorPopupMenu

5.3 Conclusion

This section has reviewed the results of dart hints and how it works. It also discussed Dart Inlay Hints plugin features. The plugin features were folding parameter type and name together for additional information, a second feature was displaying hints only on non-statically typed variables, also, only showing the button on dart files. And not on non-dart files.

Section 6: Project Conclusion

6.1 Introduction

This section will contain an epilogue for the report and for project.

6.2 Gained Experiences

This project qualified the developers and added a good amount of experiences:

- **The project qualified the project developers to develop more plugins to help the developers society.**
- **Deep understanding of how event driven IDEs works.**
- **Deep understanding of intellij idea platform which has lighten up the project developer to how good code must be written.**
- **Developed new abilities like creating solutions out of elicitation, and a skill of quick understand of codes; as project developers had to read and understand tremendous amount of codes.**
- **Understanded how a programming language compiler could be integrated into an IDE.**
- **A new way of programming thought has been developed as the project developers dived into intellij idea source code. the developers were able to grasp how complicated systems must be fragmented.**
- **Learned new languages such as XML, kotlin.**
- **Learned how to Grasp information out of poor-quality informative texts meant to be documents.**

6.3 Challenges

The developers faced few challenges but they were enormous in impact:

- **Lack of resources for reading and learning**

  Plugin development sources were only one and it was poorly-written, it was not meant to be for beginners. The Dart PSI had zero sources.

- **Extremely large platform**

  The platform extremely large and finding what developers were looking for was a tough task.

- **Extremely complicated and entangled platform**

  The platform code is complicated to the point it is impossible to grasp a single tool functionality in the platform from the couple times. And every function is composite of other functions in other classes. This mess leads the reader to get lost during reading.

- **Not Fluently coded**

A feature of great functional code is: the functions and classes names must be expressive and telling. a class name must express its role, and a function must express its functionality. This feature is not properly followed in Intellij idea source code.

6.4 Challenges Overcoming Methods

Few methodologies were used to overcome the challenges:

- **Reading a lot of source codes over and over.**
- **Testing platform methods functionalities in a code lab.**
- **Debugging and tracing the platform methods in order to understand the flow of data in the code.**
- **Searching the marketplace for similar ideas and implementations.**

6.5 Future Prospects

The developers of this project are intending to extend their work through some steps:

- **Publishing the plugin to JetBrains Market.**
- **Extending the plugin usability to older intellij idea versions.**
- **Extending the plugin usability to android studio for flutter support.**
- **Extending the hints to other type of hints rather than variable and parameters hints. Like error hints.**
- **Creating a plugin to assist developers in plugin developers.**

References

1. https://upsource.jetbrains.com/idea-ce/structure/idea-ce-6ed4281aa53e1672d2e5870243c6606bb48afc84/platform/core-api/src/com/intellij/codeInsight
   (2022/03 ~ 2022/06)
2. https://upsource.jetbrains.com/idea-ce/structure/idea-ce-6ed4281aa53e1672d2e5870243c6606bb48afc84/platform/core-api/src/com/intellij/openapi
   (2022/03 ~ 2022/06)
3. https://plugins.jetbrains.com/docs/intellij/psi.html
   (2022/03 ~ 2022/06)
4. https://plugins.jetbrains.com/docs/intellij/custom-language-support.html
   (2022/06)
5. https://docs.gradle.org/current/userguide/what_is_gradle.html
   (2022/03)
6. https://www.jetbrains.com/help/idea/gradle.html
   (2022/03)
7. Michelson M., "Event-Driven Architecture Overview", Patricia Seybold Group, 2006
8. https://plugins.jetbrains.com/docs/intellij/implementing-lexer.html
   (2022/06)
9. https://plugins.jetbrains.com/docs/intellij/implementing-parser-and-psi.html
   (2022/06)
10. https://plugins.jetbrains.com/docs/intellij/psi-files.html
    (2022/03 ~ 2022/06)
11. https://plugins.jetbrains.com/docs/intellij/basic-action-system.html#principal-implementation-overrides
    (2022/03 ~ 2022/06)
12. https://plugins.jetbrains.com/docs/intellij/syntax-errors.html
    (2022/06)
13. Larman. C. *Applying UML and Patterns*, Edition 2, 1997 Ch. 6, Pages (61-67).
14. Dr.Graf. G. (2008) "Space Systems Engineering: Functional Analysis Module", Vol. 1, NASA, available: http://mercury.pr.erau.edu/~siewerts/se310/documents/Papers-and-Specs/NASA_Functional_Analysis_Module_V10.pdf.
15. Albahari J. *C# 9.0 in a Nutshell , O'Reilly Media, Inc*, 2021 , Pages (512 – 543).
16. https://plugins.jetbrains.com/docs/intellij/gradle-prerequisites.html
    (2022/03)
17. https://plugins.jetbrains.com/docs/intellij/kotlin.html
    (2022/03)
18. https://plugins.jetbrains.com/docs/intellij/plugin-dependencies.html
    (2022/03 ~ 2022/06)
19. https://plugins.jetbrains.com/docs/intellij/plugin-configuration-file.html
    (2022/03 ~ 2022/06)

20. https://plugins.jetbrains.com/docs/intellij/basic-action-system.html
(2022/03 ~ 2022/06)

Appendix

A. This youtube video contains a starter lesson for plugin development in intellij idea : https://www.youtube.com/watch?v=cAwH_DbFrfw&ab_channel=IntelliJIDEAbyJetBrains
B. This video contains information about gradle system: https://www.youtube.com/watch?v=F3DF6bQo6jk&ab_channel=TomGregory
C. Gradle system second video: https://www.youtube.com/watch?v=KN-_q3ss4l0&ab_channel=Gradle
D. A tutorial used to grasp a little bit how intellij idea components work together: https://www.youtube.com/watch?v=9J0j-90dC60&ab_channel=JetBrainsTV
E. A post on jetbrains support team forum about inlay hints: https://intellij-support.jetbrains.com/hc/en-us/community/posts/4406715642130-Can-we-add-code-formatting-to-the-Inlay-hints-
F. A forum for support provided by jetbrains: https://intellij-support.jetbrains.com/hc/en-us/community/topics/200366979-IntelliJ-IDEA-Open-API-and-Plugin-Development