Conceptual Architecture

IntelliJ IDEA

EECS 4314
Dr. Jack Jiang
October 7th, 2019

Report Prepared by Lamport's Legion:

Paul Sison
Connor A
Aya Abu Allan
Jonas Laya
Jeremy Winkler
Damanveer Bharaj
Connor D

Table of Contents

1.0 Abstract

This report provides an overview of the conceptual architecture of IntelliJ IDEA. There are two major parts of this report, the first part provides information about the high-level architecture on IntelliJ IDEA. The second part is about the breakdown of IntelliJ IDEA's subsystems and the interaction and dependencies between them. The first part begins by providing diagrams and detailed descriptions of the high-level architecture styles and design patterns identified in IntelliJ IDEA. This is followed by explaining how Eclipse's architecture was used as a reference and how we came up with our own "Parthenon Architecture". A brief summary of the system's evolution is provided including the notable changes for each version update. The second part begins by providing a breakdown of the subsystems in IntelliJ IDEA. For each subsystem, detailed descriptions and diagrams are provided. The interactions and dependencies between subsystems are explained with the use of sequence diagrams and use cases. Finally, we conclude this report by providing the derivation process of this report, the lessons our group learned throughout making this report and the overall conclusion of the report.

2.0 High-Level Architecture

This section provides detailed descriptions of the important high-level architecture styles and design patterns identified in IntelliJ IDEA. First, we will present the conceptual architecture found from sources such as the IntelliJ documentation, as well as recorded videos of IntelliJ's CEO. Then we will discuss some reference architecture for the program, and finally, observe how the system's architecture has evolved over time and will continue to evolve.

2.3 High-Level Conceptual Architecture

Below, Fig. 1 shows the conceptual architecture that we have identified for the IntelliJ IDEA IDE. We will be discussing 3 main top-level architectures used in this project: component-based architecture, layered architecture, and product line architecture.

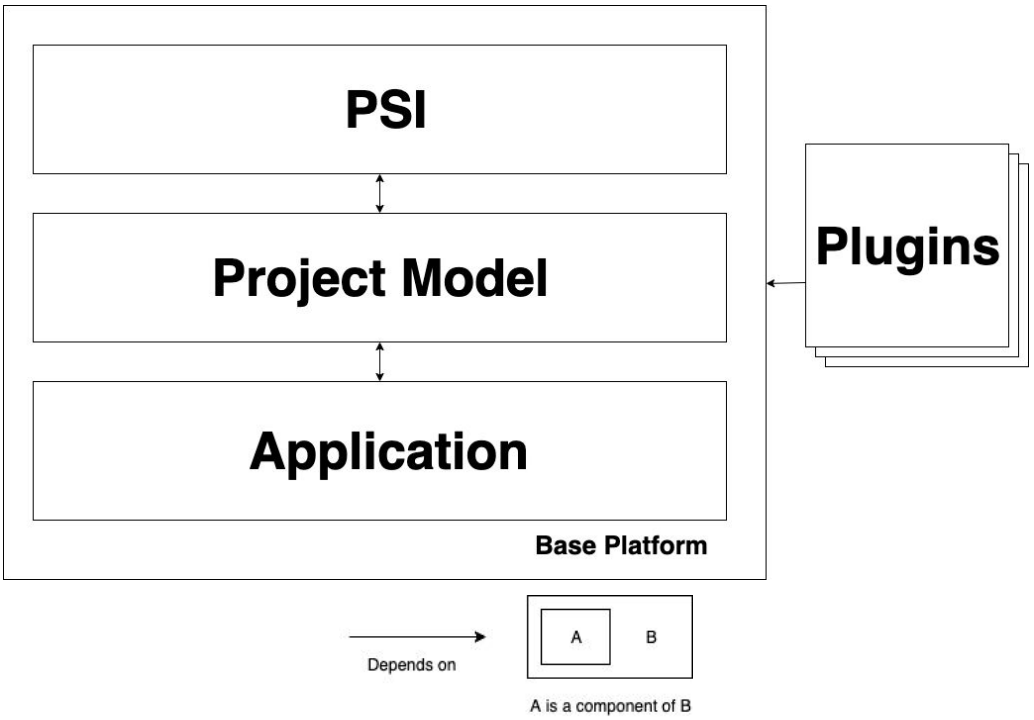


Figure 1. The architecture of IntelliJ IDEA

2.3.1 Component-Based Architecture in IDEA

Component-based architecture is the architectural paradigm that emphasizes on splitting up the functionality of the software system into their functional components. For IDEA, they have done that by separating their project into 4 main components: the application, the project model, the Program Structure Interface (PSI), and the Plugin model. These components are held together by the base platform, which provides additional functionality such as messaging between the components but are separated to create clear interfaces between different sets of functionality within the system.

As will be seen later, this is a common architectural style for other application programs like Eclipse, since it creates a logical separation of concerns of each of the parts of the application.

2.3.2 Layered Architecture in IDEA

Each of the components represents a different sub-system within IDEA and contains its own components within them. These subsystems will be discussed in detail in the later sections. In general, however, the application component includes fundamental application features such as the UI and editor components, the project model component is what creates the abstract model of a user's project within the IDE, and the PSI is what provides context to the project through the use of a parser and lexer.

The layered nature of the architecture ensures a clean interface between each of the different components. This allows developers to keep a good separation of concerns for each of the different functionalities that are needed and is often used as an effort to uphold the single-choice design principle. For example, the application layer is primarily concerned about displaying the project model, so it only needs to interact with the project. The project

model needs an additional suite of functionality above just the modelling of the project, so the PSI layer provides that.

Layered architecture is commonly used in systems that require the transmission of data (like communication networks), and although this isn't a network, it still creates a coherent flow of data between the different components.

2.3.3 Product Line Architecture in IDEA

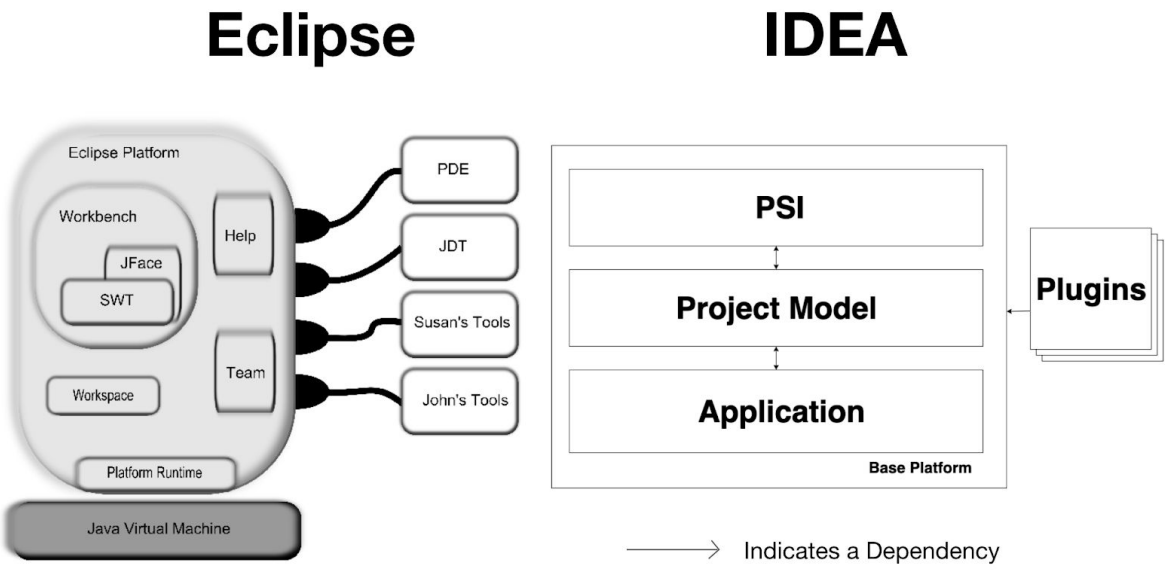
IntelliJ designed IDEA in a way that it is easily adapted to different languages. This has given them the opportunity to create over a dozen different applications built off of this platform to serve the different needs of developers.

IDEA uses "language packs" and SDKs in order to decide what the IDE needs to look for in terms of syntax highlighting, error checking, which compiler to run, and so on. By designing with this end in mind, the architecture is based on and thrives off a lively plugin library created by themselves and the open-source community, and has given them the capability to grow into many different products.

Furthermore, it has allowed them to control which of the features of the application are available in the community (free) version versus their "ultimate" (paid) version. Again, this demonstrates the use of PLA in their design.

2.2 Reference Architecture

The most similar system that we found with a detailed description of its architecture was the Eclipse IDE. Since Eclipse is another IDE that can be made to work with more languages than just Java (similar to IDEA), it has been designed in a very similar fashion. As can be seen below in Fig. 2, the Eclipse IDE also uses a component model with plugins in order to extend its functionality.



[1] "The Architecture of Open Source Software" , "Eclipse", http://www.aosabook.org/en/eclipse.html

Figure 2. Comparison between Eclipse and IntelliJ architecture

Since IntelliJ's architecture is a combination of layered and component-based, we wanted to find a memorable way to describe the architecture for new developers. Traditionally, this has been called the "toaster model", but our group saw that it very naturally lent itself to the architecture of the ancient greeks as well, and we have called it a "Parthenon Architecture" (Fig. 3).

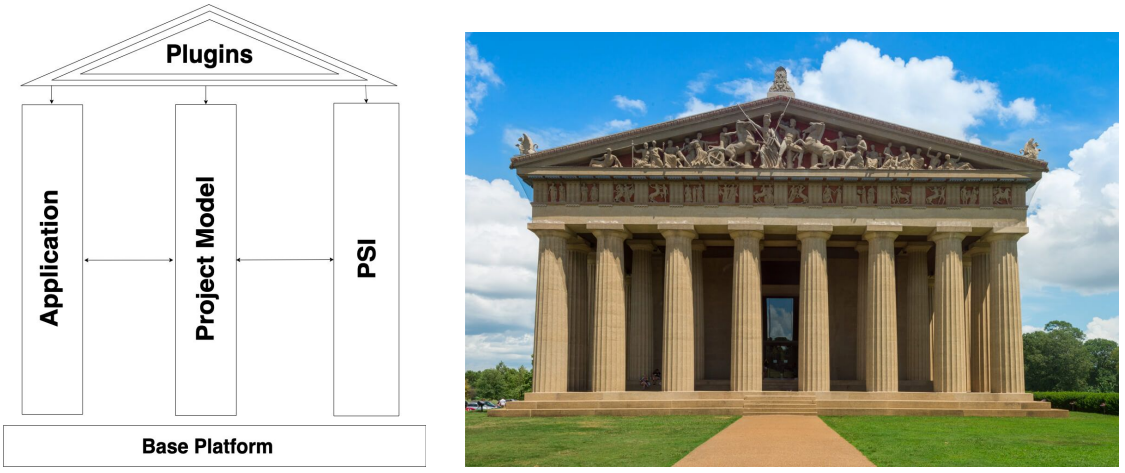


Figure 3. Comparison between IntelliJ architecture and the Parthenon (reconstruction)

2.4 System Evolution & Implications for Developers

With its first release in 2001, IntelliJ marked its spot as one of the most advanced Java IDEs in the market. The techniques used to development IntelliJ allow for continuous growth in the functionality and compatibility of the system. A very important aspect of IntelliJ's evolution is that both IntelliJ developers and the common everyday users have the opportunity and tools required to enhance the software.

IntelliJ's platform is an open-source environment provides the necessary tools required to develop and improve IntelliJ IDEA with desirable functionality. One example is in IntelliJ IDEA's Community Edition. This open-source version of the IDE is free to download and provides most of the features that everyday IDE users require. Users of the IntelliJ IDEA IDE are granted the capability to design and create plugins, contribute to bug reports, and even develop patches to the source code itself [1].

Should an IntelliJ user require added/modified functionality to their version of the local client, they can develop plugins for the IDE. Developed plugins are added to the Plugin Repository[1]. Here other users can download and install developed plugins to their local clients. Thus, allowing users to collaborate together in the development and improvement of their IntelliJ IDEs. This is a significant implication for developers. IntelliJ does not lock their users in one strict application. But provides extendibility in its services through the installation of plugins. Many plugins then become essential to a developer and are therefore used in their client at all times as if it was part of the base IDE. The ins and outs of how plugins work will be described in the Subsystems section of this report.

Furthermore, being open-source, IntelliJ users have access to IntelliJ IDEA's repository on GitHub and can fork the source code[1]. Users can then take a look into this

source code to find solutions for bug reports or locate areas where improvement is possible. Users will be able to create patches to the forked source code. This patch can provide a solution to a bug in the system, or provide a more efficient version of a utilized algorithm. The user can then create a pull request on the repository and have an IntelliJ developer take a look into it[1]. If the patch is valid and follows all of IntelliJ's coding and testing standards, the developer can create the commit to the source code[1]. As a result, IntelliJ IDEA has evolved since its initial release and will continue to evolve with the help of its community. A community/developer relationship used to create plugins and patches that add to the core functionality of IntelliJ IDEA that every user can benefit from.

3.0 Subsystems

As seen from IntelliJ IDEA's architectural analysis, IntelliJ IDEA's design and implementation consists of five main subsystems: the base platform, application layer, project model, program structure interface and plugin module. Each of these subsystems is further broken down into their own subsystems (components) to be explained in the coming sections. The unique responsibilities of each main subsystem are accomplished by their respective subsystems. IntelliJ IDEA is then the product of each main subsystem coming together through information dependency, to provide one of the richest software in the java development market.

3.1 Base Platform

3.1.1 Message Passing

Message Passing is a line of communication between processes or objects where they can send and receive data packets. This is implemented mainly with two elements: topics and message buses. Every client can subscribe to a topic on a message bus to read or send data.

3.2 Application Layer

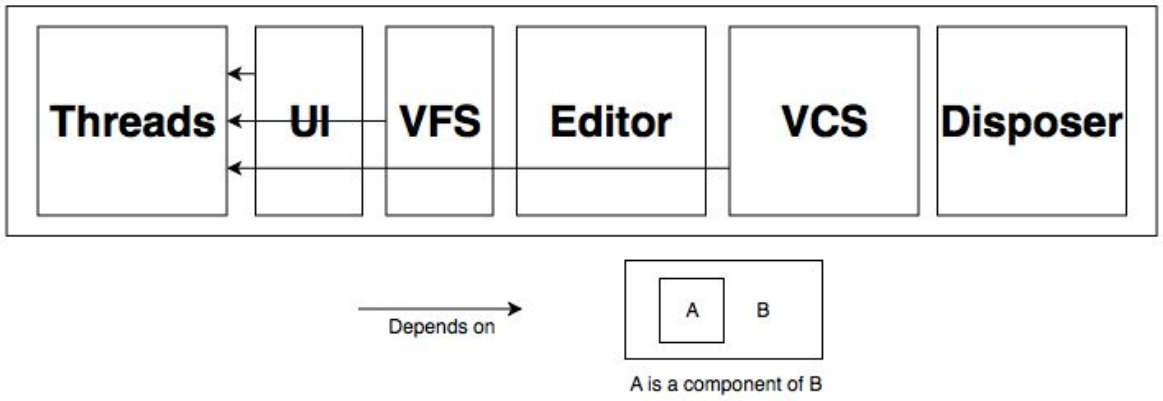


Figure 4. Application Layer including only major dependencies for simplicity

3.2.1 Threading

Threading is a vital feature of any platform that wishes to host concurrent processes. IntelliJ uses a single read/write lock for all its data structures. Reading is allowed from any thread whereas writing is only allowed from a UI thread. To maintain the integrity of every thread, modifications to the model are only permitted under write-safe contexts.

3.2.2 User Interface

The User Interface is an essential part of any software product. The IntelliJ Platform sports the typical Swing components such as toolbars, menus, an editor, file navigator, and dialogue boxes. This is also a highly customizable component of this layer.

3.2.3 Virtual File System

The Virtual File System (VFS) encapsulates most file-related activity across the platform such as tracking any modifications. Also, it simplifies the user's experience by providing a universal API for dealing with files regardless of their location. Another advantage of VFS is the implementation of "persistent snapshot". This snapshot regularly stores files from the user's hard disk that has been requested at least once for easy access. If a copy doesn't exist, the persistent snapshot will create a new one that will be shared with all future calls from the application.

3.2.4 Editor

Here is where the user interacts with the IDE the most to write, read, and edit their programs. The Editor is compatible with the running language pack and is often customized with UI plugins.

3.2.5 Version Control System

A Version Control System (VCS) allows the user to detect, manage, and merge changes made to collaborative source code. IntelliJ IDEA permits version control at the project level by associating the VCS with the root. Some VCSs that IntelliJ supports include Git, Mercurial, and Perforce.

3.3 Project Model

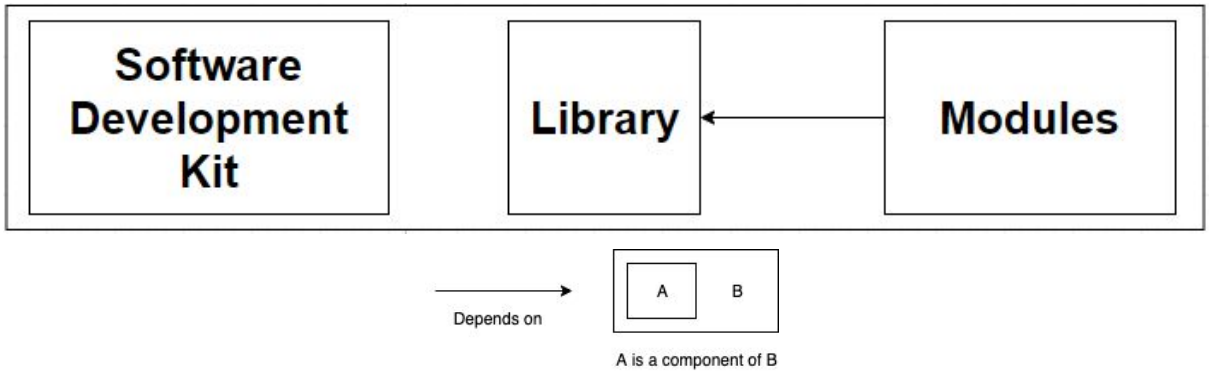


Figure 5. Components within the Project Model

The project model is essentially the description of the project. It includes everything needed to define a project. This grouping is more of a logical grouping since instead of having many dependencies within the project model, it is grouped as a set of related data which is the project. The project model contains three main components which are the SDK, the Library, and modules [1].

3.3.1 Software Development Kit

The SDK is something essential to every project and needs to be defined for each. When working with IntelliJ IDEA, the SDK used is always the Java Development Kit (JDK)[1]. The JDK consists of functionality required for writing and working with java code such as a compiler. Both the Application Layer and PSI require the JDK for various functionality. Although the other components within the Program Model don't depend on the SDK, it is still grouped here as a logical grouping since it is part of the definition of the project.

3.3.2 Library

Libraries are precompiled code required by your modules [1]. Libraries are further separated into modules and global libraries that define where the libraries are accessible [1]. Libraries are relied on by your code which is why there is an arrow from Modules to Libraries.

3.3.3 Modules

Modules are groups of functionality that can be run independently [1]. This mainly consists of source code, but also includes unit tests, build scripts, etc. [1]. This is a large portion of the project model and is often used by other submodules.

3.3.4 Relation to other subsystems

The project module is highly relied upon by other components since it contains the source code and SDK. Anything compiling-related depends on the SDK for the compiler and the modules for build scripts. Also, the lexer, parser, and virtual file system all are source code related, which means they have a dependency on the modules from the project model.

3.3.4 Find Usage Sequence Diagram

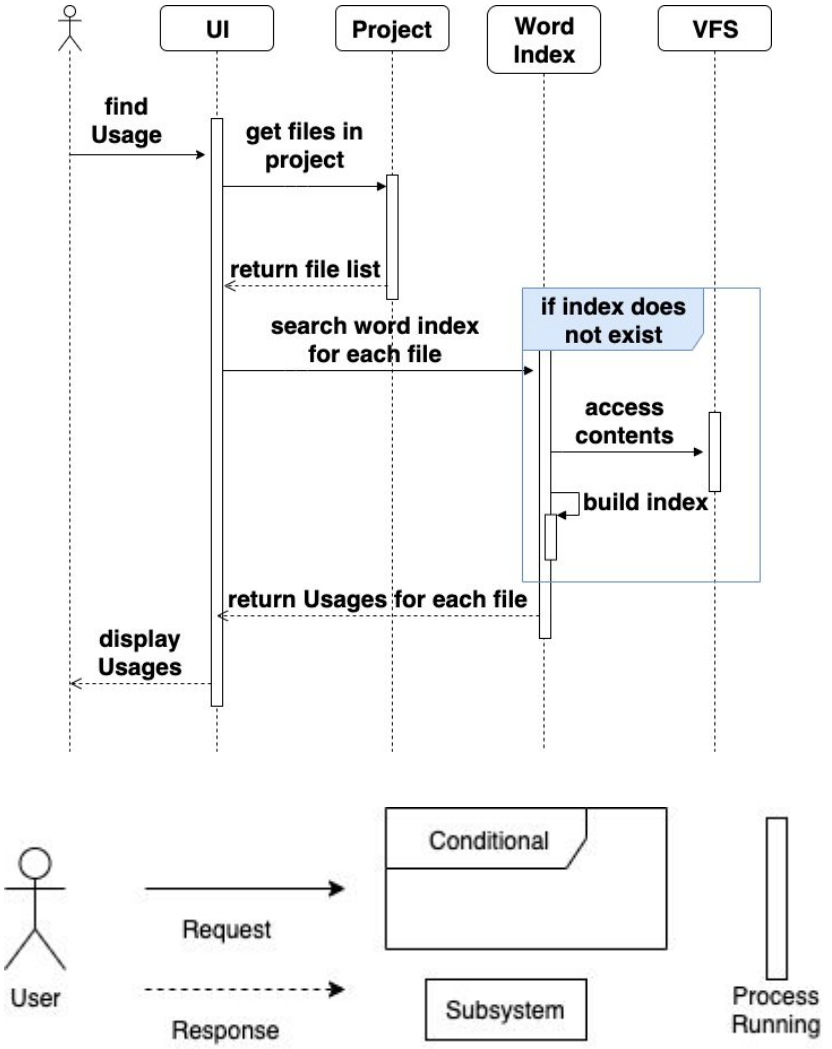


Figure 6. Sequence diagram of find usage use case

This sequence diagram shows how the project model is used by other submodules. In this diagram, the find usage request is started in the UI, which then communicates with the project model to get the list of files. This list of files could be libraries or modules. Then it moves on to get or build the index and display the usages to the user.

This diagram shows a common interaction between the other submodules and the project model. Similar to here, the other submodules depend on the project model to get the data (files, compiler, etc.) from the project model, and then process it further within their own submodules.

3.4 Program Structure Interface (PSI)

The Program Structure Interface (PSI) is what transforms IntelliJ from a text editor and a file/project manager into a complete programming environment. PSI enables features such as syntax and error highlighting, code insights, code completion, quick fixes, and other refactoring tools. To perform the said features, PSI relies on several components—the

Lexer, Parser, PSI trees, Stub indexes, and References. These components work together in a pipeline fashion to make a user's code searchable, traversable and identifiable upon request.

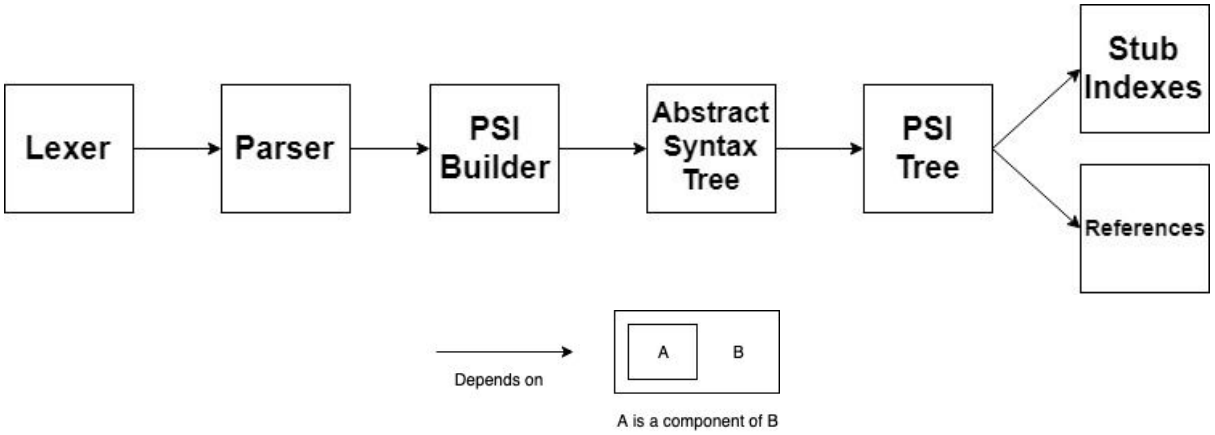


Figure 7. Pipeline processing between PSI components

PSI begins its process by first acquiring the user's source code contents via FileViewProviders given by the Project model instance to which the source file belongs. It passes the code to the lexical analyzer, also known as the lexer, which breaks the code into tokens. What counts as tokens are defined for each language. The token stream produced by the lexer is then passed to the Parser for further syntax and semantic checks.

The Parser uses a PsiBuilder instance to hold and mark the stream of tokens into literals and expressions if possible. An Abstract Syntax Tree (AST) is built from the PsiBuilder stream where tokens found between a pair of markers are grouped as sibling nodes under the same parent node.

Once we have the complete AST for the source file, a PSI tree is built on top of the AST with nodes that are instances of the PsiElement class. PSI elements, depending on the context, can then be serialized and indexed as stub indexes to be used for optimizing the IDE's search features. References are also produced from PSI elements when the semantics of the programming language or its APIs identifies the PSI element as something that can be resolved to a corresponding declaration within the codebase. PSI references are crucial in implementing code insights, code completion and other refactoring utilities.

3.4.1 Control Flow Example (Sequence Diagram)

Error highlighting is one of the most commonly implemented features in an IDE. The following sequence diagram shows how IntelliJ performs error highlighting on different levels whenever the PSI builds or reparses the PSI tree.

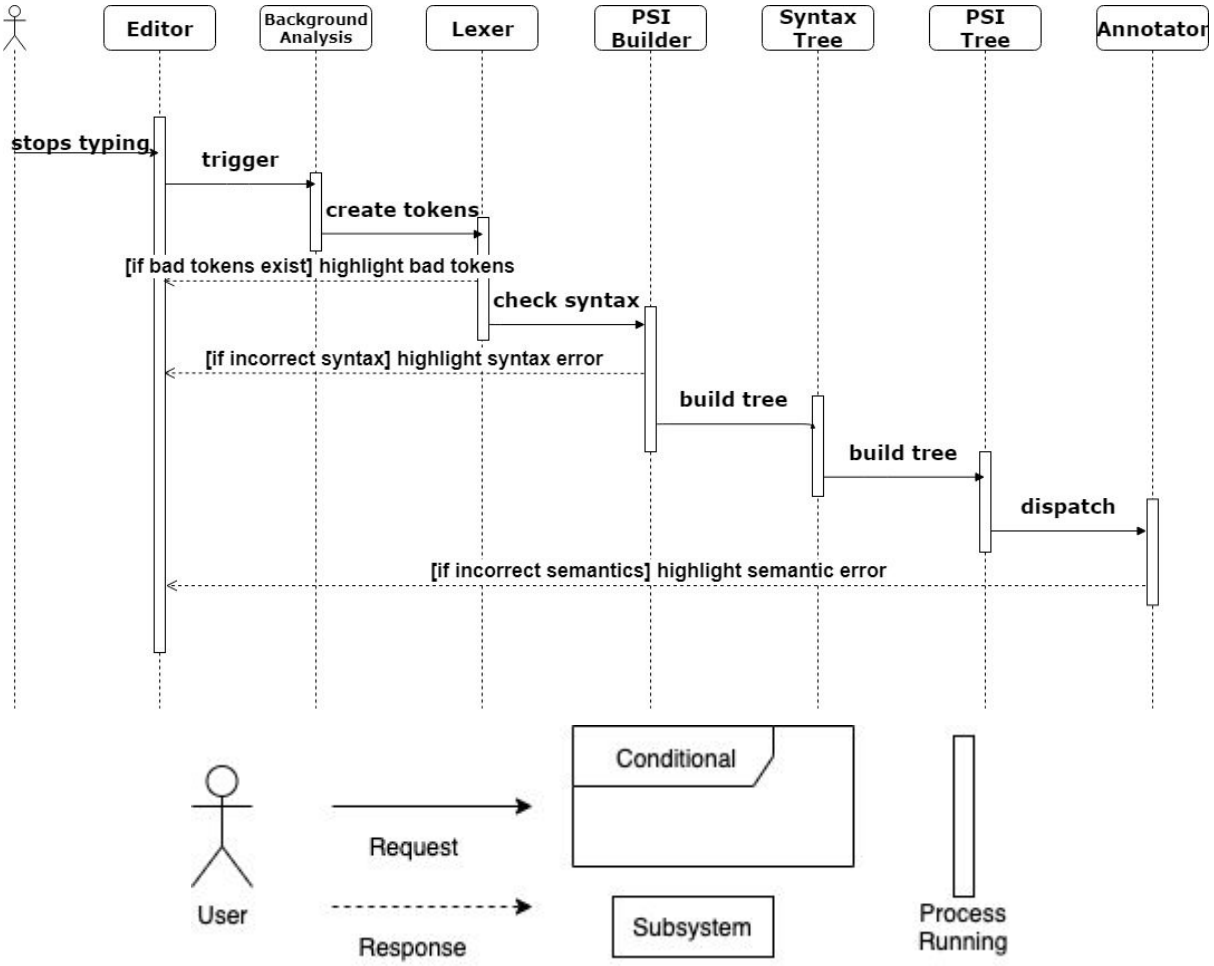


Figure 8. Sequence diagram of an “error highlighting” use case

3.4.2 Concurrency in PSI

Some of the concurrency within IntelliJ happens at the PSI level where operations to the PSI file require a read/write lock. A PSI file is a representation of a source code file as a hierarchy of PSI trees and elements. It is information that gets stored locally on the disk together with the user’s source code. As the user continuously makes changes to their code, the PSI trees constantly get reparsed and rebuilt. And since a PSI file only has project scope, that is, the same local source code can be represented as different instances of a PSI file between projects, storing the changes made to its PSI trees require the acquisition of a write lock, and then the lock’s release when the write operation is finished.

3.5 Plugin Model

Plugins are essentially tools that provide additional functionality to a system. There are four most common types of plug-ins on IntelliJ IDEA. Custom language support, framework integration, tool integration and user interface add-ons. Custom language support provides basic functionality for working with a particular programming language. Framework integration provides an option to use framework-specific functionality directly from the IDE. Tool integration makes it possible to manipulate third party tools and components directly

from the IDE without switching contexts and user interface add-ons apply various changes to the standard user interface of the IDE.

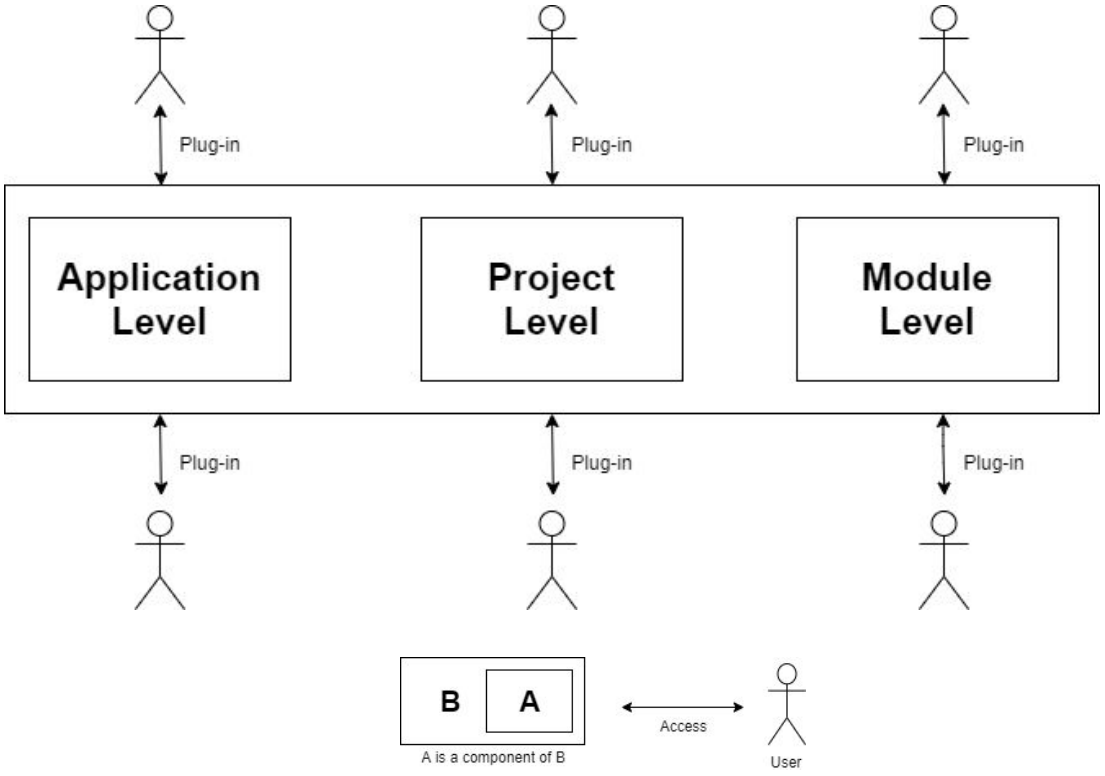


Figure 9. The Plugin Model with Components

There are three components of the plug-in model: Application level components, Project level components and Module level components.

Application Level Components:

- Plug-ins in the application component are created and initialized when the IDE starts up. They can run either independently or depend on other application-level components.

Project Level Components:

- Plug-ins in the project component are created for each project instance in the IDE. They can run either independently, depend on the application level components or depend on other project-level components.

Module Level Components:

- Plug-ins in the module component are created for each module inside every project loaded in the IDE. They can run either independently, depending on the project level components, depend on the application level components or depend on other module-level components.

Users can create their own plug-ins using Gradle and DevKit and publish to JetBrains plug-in repository so that other users can install it, represented by bidirectional arrow in Figure 8. The relation between the plug-in model and the users represents the repository style architecture. In reference to the “Parthenon Architecture” in figure 3, the plug-in model acts as the roof of the whole architecture. This analogy implies that the plug-in model interacts with other subsystems by providing additional functionality to each of the subsystems. For example, custom language support plug-ins include lexical analysis, syntax formatting and formatting which extends the functionality of the program structure interface (PSI).

4.0 Derivation Process

The two main sources that were used to extract the conceptual architecture were the Official Jetbrains Documentation on IntelliJ as well as a presentation done by Dmitry Jemerov, the Development Lead for IntelliJ IDEA at Jetbrains back in 2008. These sources were instrumental in learning the system’s architecture since they rounded each other out in a lot of ways. They also had some holes in their combined coverage of the application, this will be discussed later.

The first source the group stumbled upon was the official documentation [1]. As any engineer finds when they read documentation, the writing was pretty dry, and it was hard to read a page describing a subsystem and understand where it lies in the grand scheme of the application. The structure of the documentation helped us understand where some of the systems lie within the overall architecture, but it was tough to see what the dependencies were like across the system. This was furthered by a complete lack of high-level diagrams across the documentation, making the dependencies even harder to find.

The second source, the Jetbrains presentation [2], was found later into the project, and it really helped the group see what was going on in the system. The beauty of finding a half-hour presentation like this about the software’s architecture is that if the Lead developer thinks a component is important, he includes it in the presentation. If it isn’t important, he leaves it out. Not only does the presentation provide a “highlight reel” of important and relevant systems, but he also broke the systems down in a much more casual language than what you find in the official documentation. This helped the group see what functions of the different systems were actually important to the architecture, and identify implementation details that needed to be left out.

The Jetbrains presentation was far from perfect though, and its problems were accentuated by similar issues in the official documentation. As discussed before, the official documentation had no diagrams that were relevant to the system’s architecture, only some basic tree diagrams that explained some low-level services. The presentation had NO diagrams, just some bullet points with text. For the most part, this issue could be worked around. However, problems arose when:

1. Identifying sub-system dependencies
2. Identifying the system's architectural styles

Now, with any complicated system like IntelliJ, item (1) is generally difficult. However, even a single high-level diagram could have straightened out the group's understanding within minutes. Yet, the group was stuck meeting for long hours, making a series of educational guesses, assumptions and votes as to what made the most sense from the concrete information they had gathered.

Item (2) was almost explained in one of the first slides of the Jetbrains presentation [2]. "Component-based architecture" was one of the main points on one of the first content slides. However, this architecture is never again referenced in the presentation, and the page in the documentation is grayed out, suggesting it is either incomplete, incorrect or it may not exist. Components are listed as instrumental to the architecture of the system, but throughout our sources, they are never explained. Assumptions were made that Eclipse shares a similar reference architecture, and some better explanations were once found for their model. However, these were completely ignored later on since there was no guarantee that their architectures are similar.

5.0 Lessons Learned

The first thing that the group learned was not just about software architecture, but the software industry as a whole. This was that Java Swing is still being used in both the Open Source Community as well as for Enterprise applications. With the dominance of web applications in the past decade, it's refreshing to see that there is a market for a fourth year's "expert" abilities to develop GUIs in Java Swing from our first year. It's also a relief to see that in such a fast-moving industry, some older architectures and frameworks can stand the test of time, so long as they keep improving on themselves.

The next learning item was just how bad the Software Industry's documentation really is. In a class where nearly every other slide is a diagram, and to a degree where most projects involve software diagrams, you'd expect that an Open Source platform like IntelliJ would have some half-decent diagrams in their documentation. The overall consensus was that there were nearly no useful diagrams that were found in the building of this conceptual architecture, which is equally as disappointing as it is frustrating.

The last lesson we saw was a reminder of why we learn the things we learn in class. In class when discussions like how the Linux Kernel is organized, or how a compiler works come up, typically students don't see that information being relevant. It's not that they don't display the course concepts well, its that the amount of work available on these types of projects is fairly low, or it isn't lucrative. The OS market is mostly stagnant / monopolized, and where it isn't, its open source. Compilers on top of that are also typically Open Source. However, IntelliJ had some very strong connections to the architecture of a compiler, and the way Linux's architecture is structured helped the group visualize what was going on with IntelliJ. Each of these architectures had a strong role in extracting IntelliJ's

architecture, which helped solidify not only our understanding of these other architectures but their underlying architectural styles.

6.0 Conclusion

In the overall architecture, there are four main submodules which are the application, project model, program structure interface, and plugins. The application, project model and PSI are all part of the base platform and the plugins connect to whichever part they need depending on the functionality provided by the plugin. Together these submodules have three main architectures. The product line architecture is shown by the fact that all of IntelliJ's IDEs use the same platform and just change the plugins to work for different languages. The layered architecture is shown by the communication between the submodules. The component-based architecture shown by the separation of all the submodules into the four larger submodule groups. There is also the Parthenon architecture which is similar to the toaster model, but better describes the fact that plugins are extra functionality on top of the other components.

7.0 References

1. JetBrains, "IntelliJ Platform SDK DevGuide," *JetBrains IntelliJ Platform SDK*, 12-Mar-2019. [Online]. Available: https://www.jetbrains.org/intellij/sdk/docs/welcome.html

2. IntelliJ IDEA Architecture and Performance, GoogleTechTalks, Dmitry Jemerov, 2008, Youtube Video