Concrete Architecture

IntelliJ IDEA & PSI Subsystem

EECS 4314
Dr. Jack Jiang
November 4th, 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

The goal of this assignment is to identify the concrete architecture of the IntelliJ IDEA IDE and compare it against its abstract counterpart. First, each subsystem of the high-level concrete architecture is investigated for discrepancies; whether the functions it encompassed in the abstract architecture are correct and its dependencies with other subsystems. Next, the report takes a detailed look into the Program Structure Interface (PSI) where each discrepancy discovered was studied using a reflexion analysis. Also a description of the design patterns encountered and their validity is included. A rundown of the derivation process is also provided where we discuss our approach for extracting the concrete architecture. Lastly, we reflect on the lessons learned from such process and draw conclusions on the significance of differences from the concrete to abstract architecture and what they may imply.

2.0 Introduction & Overview

This report provides an in-depth analysis of the derived concrete architecture of the IntellIJ IDEA IDE, with a key focus in the Program Structure Interface (PSI). Through the investigation of source code and the utilization of reverse engineering technology, the implemented IntellIJ IDEA IDE architecture was unveiled and documented into various key sections. An initial high-level architecture was derived in order to provide an understanding of how the overall system was structured. This includes the interactions and dependencies between each component of the system, along with the component, layered, and product line architectural styles used to develop the software. Our attention was then placed into a key component of the high-level architecture, the Program Structure Interface. Utilizing similar derivation methods, the PSI was unmasked, and a concrete architecture developed.

The developed high-level and PSI concrete architectures were then compared to their respective conceptual architectures. These conceptual architectures were developed and reported in our previous assessment titled “Conceptual Architecture: IntelliJ IDEA”, by Lamport’s Legion [3]. Placing our view back to the current report, a reflexion analysis was performed on the models. Each respective concrete and conceptual architecture were placed side-by-side and evaluated for divergences and absences (gaps). Divergences can be defined as dependencies in the concrete architecture which were not present in the conceptual architecture. While absences are the opposite, dependencies in the conceptual architecture which are not present in the concrete architecture. As such, each gap was studied in order to determine the rationale behind the differences.

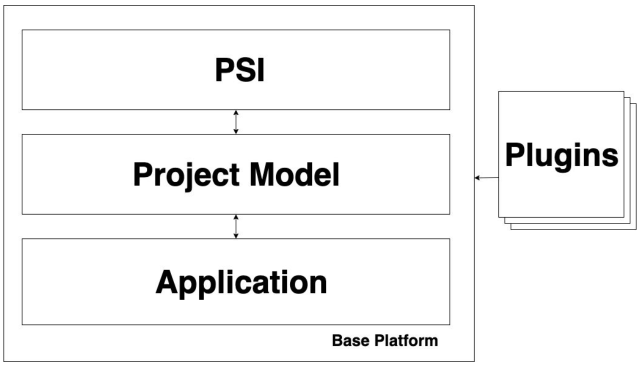
This report then signifies several areas of interest and notable use cases discovered while studying the concrete architecture of IntelliJ IDEA IDE. Noteworthy design patterns such as the Visitor and Observer design patterns were discovered to be used in the PSI’s implementation. It is understood that much of the PSI’s functionality comes as a result of these design patterns.

Finally, our approach on acquiring and studying the data presented throughout this paper are presented. An in-depth description on the derivation process are disclosed indicating the tools and result breakdown used to develop the concrete architecture. With many barriers and challenges presented along the way, the report notes lessons learned about performing such tasks, and about the complexity that is software architecture in reality.

3.0 High-Level Concrete Architecture

3.1 Concrete Architecture and Main Components

The overall structure of the concrete architecture was very similar to what we found in our conceptual architecture. One of the main changes found was that, instead of the base platform being a grouping of the PSI, project model, and application, it was in fact its own component.



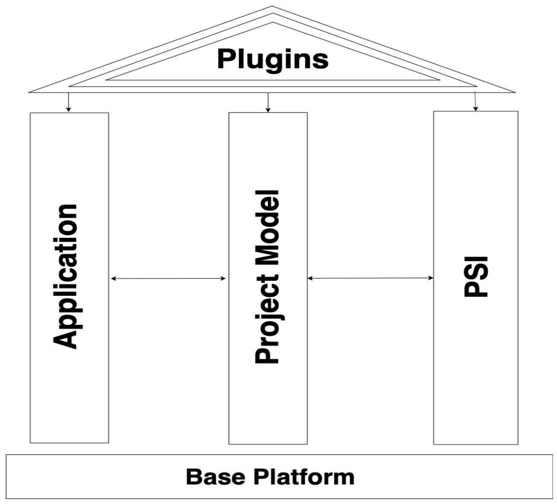


Figure 1: High-level conceptual architecture **Figure 2**: Parthenon Architecture

From this conceptual architecture (fig. 1), we related it to some architectural styles such as the Parthenon architecture seen above (fig. 2), which is very similar to the toaster model. When extracting the concrete architecture, we started with the same idea and built off of it to create the concrete architecture shown below.

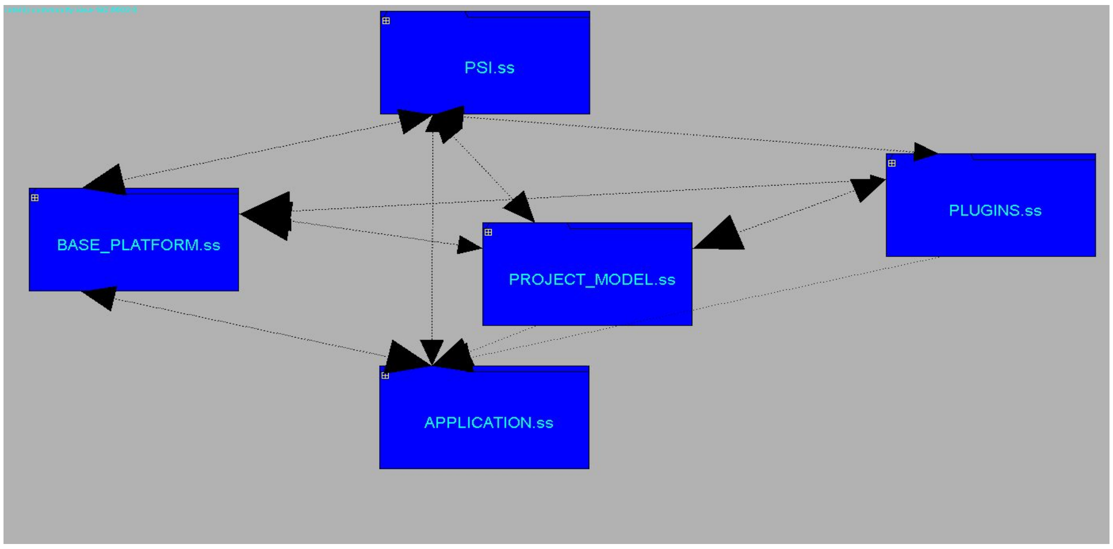


Figure 3: LSEdit High-level concrete architecture

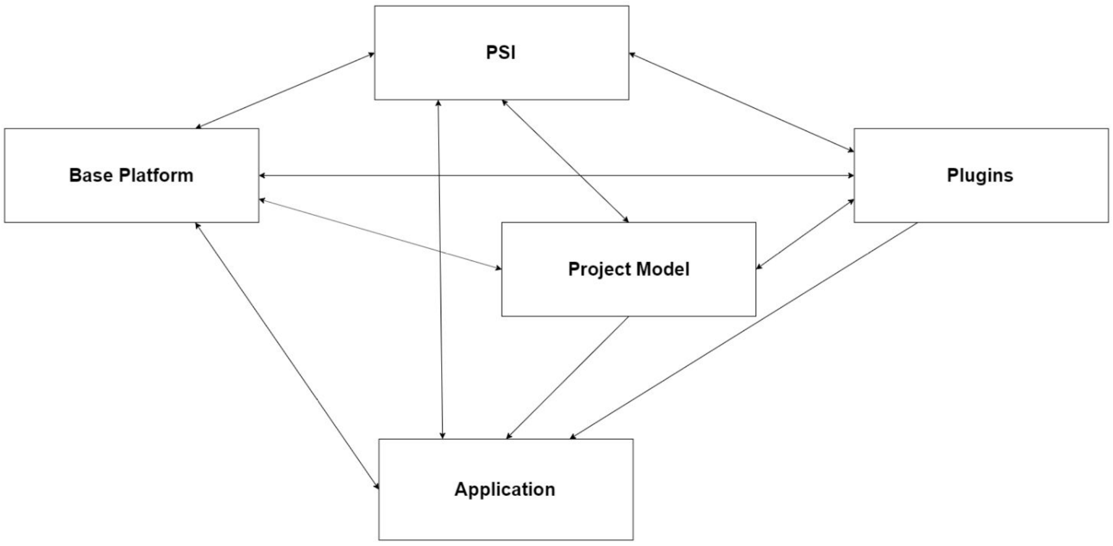


Figure 4: High-level concrete architecture

This architecture is very similar, but with the base platform extracted as its own sub-module. Using this diagram we can still relate it to the Parthenon or toaster model seen in **Figure 5**.

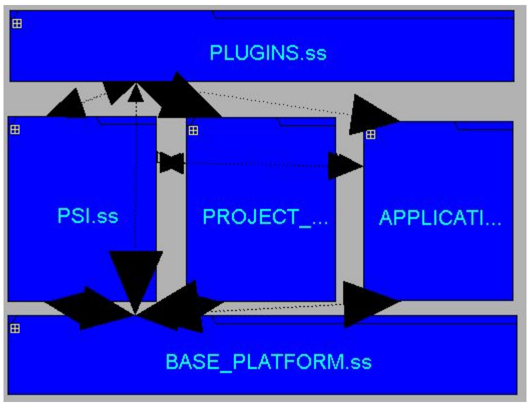


Figure 5: Toaster Model diagram of concrete architecture

3.1.1 Base Platform

The base platform being its own component is the main difference found in our concrete architecture. The base platform contains most of the basic functionality of the system and includes many extra language support which is needed for any version of IntelliJ. Some of this extra support includes python, regex, json, and xml. The messaging and threading components that were previously in the application layer were also moved to the base platform. Due to this core functionality being added to the base platform, it became depended on by every other main subsystem in IntelliJ. Another thing that changed was the base platform's dependency on every other subsystem as well. In our conceptual, we thought the application layer would have a lot of functionality needed by other components, but that it wouldn't depend on other components as heavily. However, this interpretation changed when seeing the concrete architecture as it also depends on everything else.

3.1.2 Application

The application was changed from being kind of a general component that contained lots of functionality, to mainly containing UI components. So, now that the application layer lost most of its subsystems, it focuses more on the UI of IntelliJ. Although most of the application's functionality was stripped and moved to the base platform, the dependencies on the application did not change much. The dependency on the project model was removed, but dependencies on the plugins, PSI, and base platform were added.

3.1.3 Project Model

The project model stayed relatively similar to our concrete architecture. The project model is the representation of the user's project and contains all necessary information related to that project. This information includes the SDK, libraries, and modules [1]. The SDK is something necessary for every project and includes functionality for working with code such as the compiler. Libraries are extra pieces of code needed by your project which you imported. These libraries are split into different levels which define the visibility level of that library [1]. Finally, the modules represent anything that can be run, tested, or debugged [1]. The main component of the modules is the source code, but it also includes build scripts, unit tests, etc [1].

3.1.4 PSI

The PSI, which is the component we are studying in this report, also is relatively similar to our conceptual architecture. It has a few main components including lexer, parser, indexes, PSI tree, etc. The overall goal of the PSI is to break the source code and libraries used down into smaller chunks of information which can be easily dealt with and understood by the computer. Breaking down the code into a tree of understandable components allows for lots of useful functionality specific to an IDE such as IntelliJ. In terms of dependencies, the PSI stayed relatively similar, but dependencies on the application, base platform, and plugins were added.

3.1.5 Plugins

The plugins are the main way the IntelliJ platform grows. Plugins are a way for developers to add additional features to the system without having to know too deeply how everything works. Our Parthenon architecture idea was created from this since the rest of the system is kind of like a base to hold up the plugins which are just added in on top of the system. The plugins are interchangeable and are the way IntelliJ uses the product line architecture to make different IDEs for different languages by exchanging the plugins used. One large misconception in our conceptual architecture was that the other components were only depended on by the plugins, but the plugins were not depended on by the other subsystems. This was flawed since once we created the concrete architecture, we realized that the other components also heavily depend on the plugins.

3.2 Concurrency and Performance Critical

As stated before, the messaging and threading subsystems of IntelliJ were moved from the application to the base platform. These subsystems are the main components that allow Intellij to run concurrently. Many features in the base platform require the threading to ensure it doesn't interfere with the user interacting with the system. The PSI has a similar issue, where many of the benefits of using an IDE such as code completion require parsing and interpreting the user's actions using the functionality of the PSI. Since this is run at the same time as the user is interacting with the system, then it must be done concurrently through the threading component of the base platform.

Now that most of the functionality from the application was moved to the base platform, the application has become less of a concern, but that means the base platform now is concerning performance-wise. As an IDE, there is a lot of functionality that comes with it, but lots of this does not require heavy use of the computer. As such the performance-critical components are less of an issue. In this system, things like parsing, lexing, and indexing are likely to be some of the most computation heavy when opening a new project since all files must be parsed, lexed, and indexed.

4.0 PSI Concrete Architecture

4.1 Architectural Styles

There are many dependencies within the PSI concrete architecture, however as seen in **Figure 7**, most of these dependencies were completely different than our conceptual understanding of the subsystem.

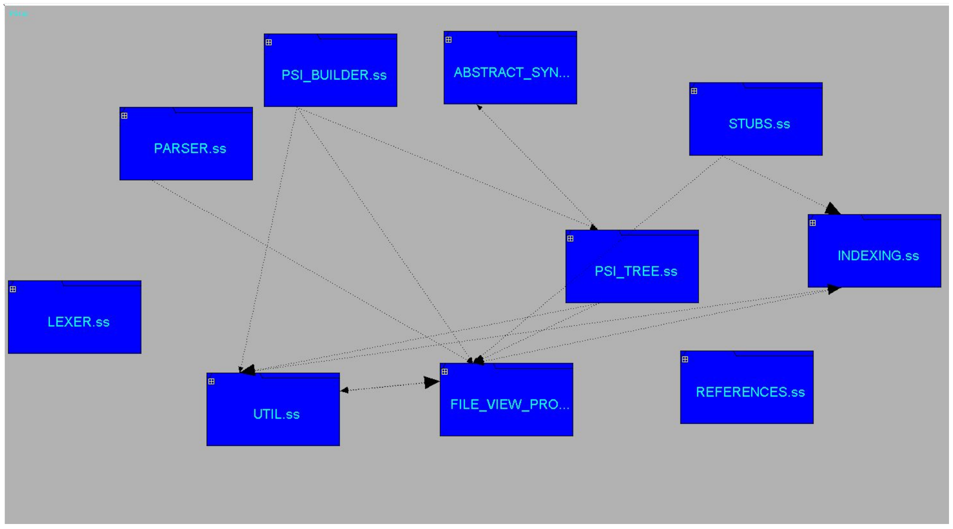


Figure 6: LSEdit Concrete PSI Architecture

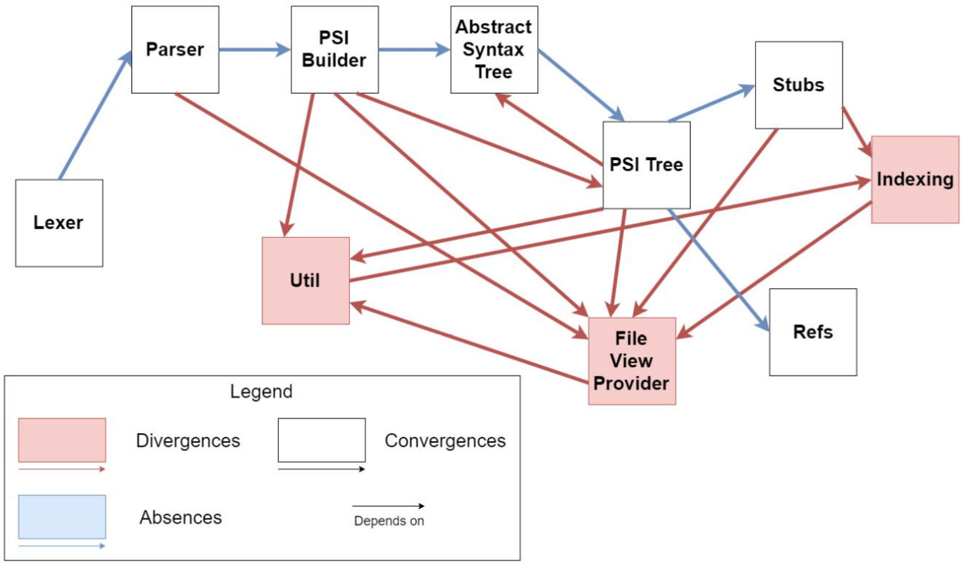


Figure 7: Concrete PSI Architecture

In our conceptual architecture, we found that a pipeline architecture was needed for the PSI, however, after looking at our concrete architecture we see that every dependency we found in our conceptual architecture was removed. Along with the removal of old dependencies, came many new dependencies from which we can see a new architectural style appearing, which is the repository style. As seen in **Figure 7**, the File View Provider is depended on by most subsystems within the PSI. This is due to the fact that many of the PSI components rely heavily on the source code as it deals with understanding the source code. Each of the subsystems needs some way to access these files, so the File View Provider is that point of access to the source code which creates a repository style architecture.

4.2 Discrepancies between Concrete and Conceptual

During the last assignment, the workings of the PSI was summarized into the following conceptual architecture:

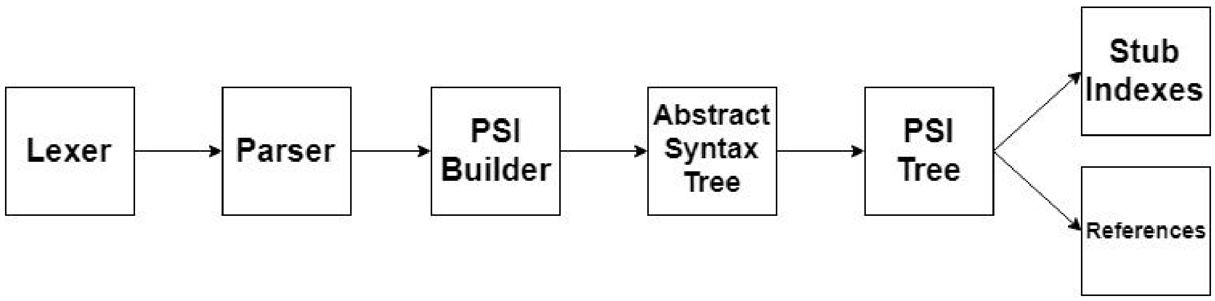


Figure 8: PSI Conceptual Architecture

The diagram above is intuitive when looking at the flow of lexer tokens through the PSI subsystem. Once the lexer generates lexer tokens, they get passed through a variety of different components and processes until they reach the PSI trees. On the PSI trees, they exist as components of stubs and references.

However, when we look at the concrete architecture, we see that this isn't the complete picture of the PSI's architecture.

4.2.1 Dependency Absences

We can see above that there isn't a clear relationship between the Lexer, Parser, References and the PSI. These components act as separate entities within the system, with other subsystems facilitating their communication.

These highlights indicate our Conceptual Architecture's **Dependency Absences**. These are the dependencies that were drawn in our Conceptual Architecture that don't exist in the concrete system.

In **Figure 9**, you can see that there are many more dependencies of the PSI components when you broaden your viewing scope. When we include more subsystems to our view, we can see more dependencies with the PSI throughout the IntelliJ platform than the conceptual architecture accounted for. If one looks closely, it can be seen that dependencies exist between the Lexer, Project Model, Platform Core and Parser. These dependencies are bijections between our conceptual PSI components, the Core and the Project Model. Even though LSEdit doesn't display a sequence of dependencies from the Lexer to the PSI Builder, we imply that there are indirect dependencies that act similar to the conceptual architecture. These indirect dependencies exist as dependencies to and from the Project Model and the Platform Core.

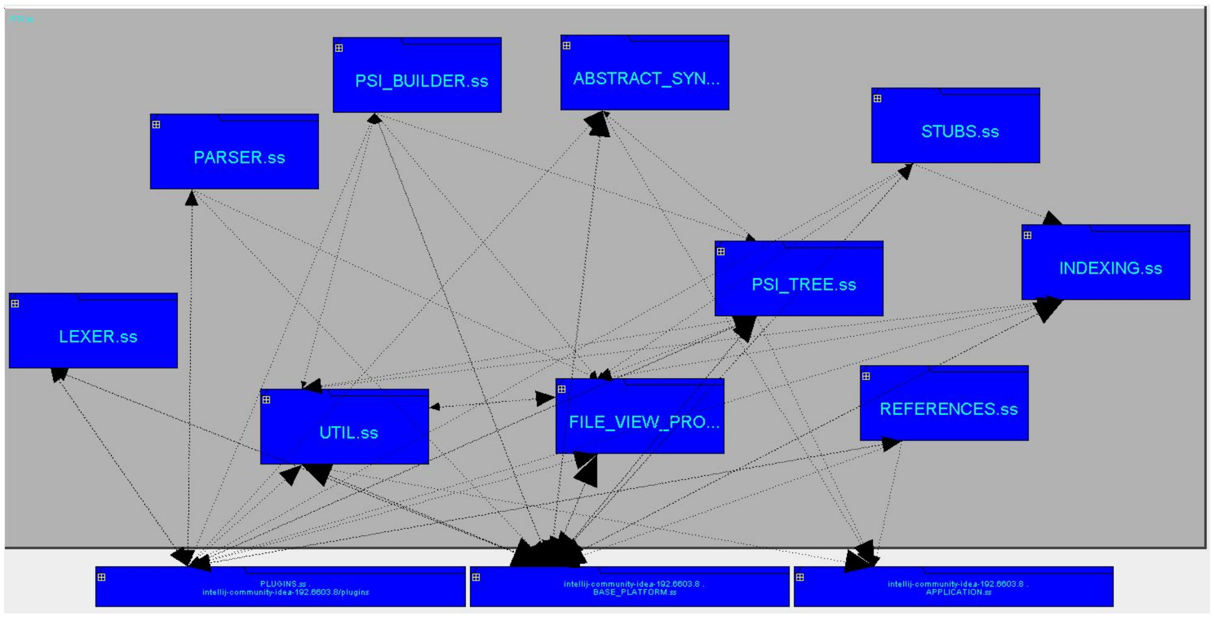

PSI_BUILDER.ss
ABSTRACT_SYN...
STUBS.ss
PARSER.ss
INDEXING.ss
PSI_TREE.ss
LEXER.ss
REFERENCES.ss
UTIL.ss
FILE_VIEW_PRO...
PLUGINS.ss . intellij-community-idea-192.6603.8/plugins
intellij-community-idea-192.6603.8 . BASE_PLATFORM.ss
intellij-community-idea-192.6603.8 . APPLICATION.ss

Figure 9: LSEdit Concrete PSI Architecture

Another big dependency absence is the References subsystem. It seems that references are language-related constructs that are provided by the Application layer to the PSI Tree. We know references are used by the PSI tree because their involvement is described in the official documentation. The relationship between the two components though is much more complicated than our Conceptual Architecture described.

4.2.2 Component Divergences

A **Component Divergence** is a component that is found in the concrete architecture that wasn't in the conceptual architecture. These are components that are important to the PSI that weren't initially accounted for.

There are 3 main component divergences that can be found between the two architectures:

1. Indexing
2. Util
3. File View Provider

The **Indexing** subsystem is the subsystem used by IntelliJ for finding and organizing different data and tokens in the PSI trees. This is the main interface the platform uses to access the PSI.

The **Util** subsystem is a set of utility classes and systems that are used across the platform. The Indexer is a component of Util, thus is included in the PSI

The **File View Provider** is the interface that other subsystems use to gain access to a given PSI tree. Being such an important part of the PSI, it makes sense to include it in the PSI architecture.

4.2.3 Dependency Divergences & Reflexion Analysis

Divergences from the Dependencies in the Conceptual Architecture are the dependencies that were missed in the Conceptual model. In most systems, these tend to be bijective relationships on previous dependencies, as well as dependencies with divergent components.

PSI Builder -> Util

Low-level code entity	import com.intellij.openapi.util.UserDataHolder;
Who added the dependency	Github User: nicity
When was it modified	June 27, 2006
Why was it added	Using the UserDataHolder class helped deal with a JSP 2.0 related error

PSI Tree -> Abstract Syntax Tree

Low-level code entity	import com.intellij.lang.ASTNode;
Who added the dependency	Github User: donnerpeter
When was it modified	Feb 7, 2018
Why was it added	Fixed a unit test

PSI Tree -> File View Provider

Low-level code entity	final FileViewProvider viewProvider = file.getViewProvider();
Who added the dependency	No username provided
When was it modified	March 1st, 2006
Why was it added	Helped eliminate getNode method

Indexing -> File View Provider

Low-level code entity	import com.intellij.psi.PsiFile;
Who added the dependency	Eugene Zhuravlev
When was it modified	March 10, 2009
Why was it added	Introducing new Indexer functional

4.3 Noteworthy Design Patterns

4.3.1 Visitor Design Pattern

The PSI subsystem makes extensive use of the visitor design pattern to provide a rich and expandable set of operations which are loosely coupled from different languages that the platform supports.

The visitor pattern separates the operations that a developer may wish to perform from the elements that will be used in the operation. This allows developers an excellent way to expand the set of operations required throughout the development process, as well as if they are required to expand the operations in the future. In IntelliJ, the two main interfaces that are used to create this design pattern are the PsiElement and PsiElementVisitor.

The visitor classes are passed into the accept() function of the Element, and depending on the element, the corresponding "visitElement()" implementation will be called. This allows for a consistent interface between the PSI subsystem and the rest of IntelliJ across different language packs. The primary example (and one that we have included a sequence diagram for later) is to use visitors as a way of navigating through a PSI File and visiting all of the elements within it.

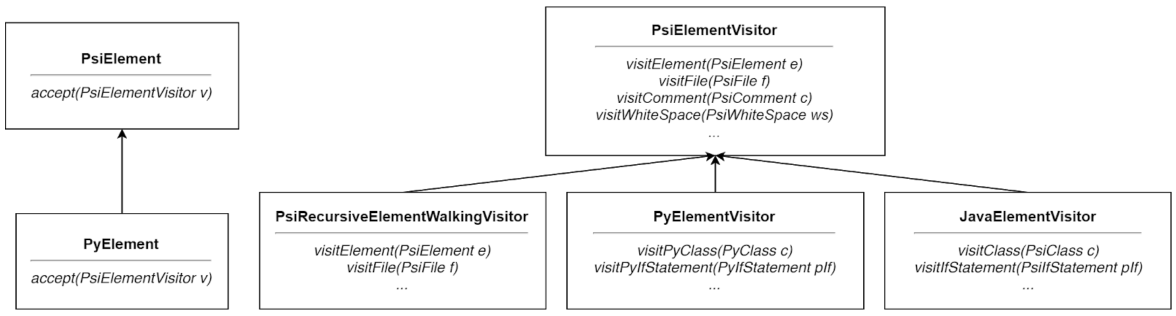

PsiElement
accept(PsiElementVisitor v)
PyElement
accept(PsiElementVisitor v)
PsiElementVisitor
visitElement(PsiElement e)
visitFile(PsiFile f)
visitComment(PsiComment c)
visitWhiteSpace(PsiWhiteSpace ws)
...
PsiRecursiveElementWalkingVisitor
visitElement(PsiElement e)
visitFile(PsiFile f)
...
PyElementVisitor
visitPyClass(PyClass c)
visitPyIfStatement(PyIfStatement pIf)
...
JavaElementVisitor
visitClass(PsiClass c)
visitIfStatement(PsiIfStatement pIf)
...

Figure 10: Visitor Design Pattern

4.3.2 Observer Design Pattern

The observer design pattern is also used heavily throughout the entire IntelliJ platform for event subscriptions and notifications. The PSI also includes an observer pattern implemented here for developers to watch for changes made within the PSI tree.

Within the PSI subsystem, the Event and Listener classes shown below (PsiTreeChangeEvent and PsiTreeChangeListener) are both implementations of the Java Observer infrastructure. They provide the necessary implementations for developers to watch the PSI tree; however, as of the version we are working with, this infrastructure is currently recommended to be used as little as possible by the developer as found in his comments within the source code:

PsiTreeChangeEvent
getOldValue
getNewValue
...
PsiTreeChangeListener
beforePropertyChange
beforeChildrenChange
...

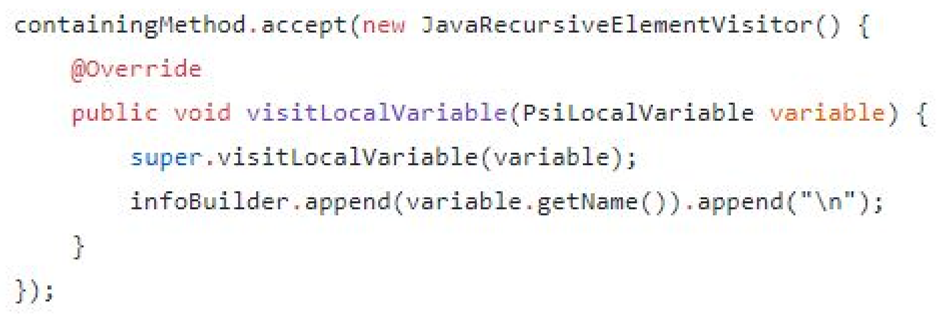
Figure 11: Processing of PSI events

```
Try to avoid processing PSI events at all cost! It's very hard to do
this correctly and handle all edge cases.

There's no specification, and the precise events you get can be
changed in the future as the infrastructure algorithms are improved
or bugs are fixed.

To say nothing of the fact that the precise events already depend on
file size and the unpredictable activity of garbage collector, so
events in production might differ from the ones you've seen in test
environment.
```

Our group found this to be particularly interesting - even though the project contains certain infrastructure (and thus from an architectural point of view, is fulfilled), in some cases such as this the implementation may be too difficult or rely on too many other factors to perform consistently.

4.4 Use Cases

A good example use case here would be for the developer to make a call to the PSI tree through the usage of a visitor. Below is a code snippet and sequence diagram demonstrating how a developer might access all of the local variables in a particular file. This example is taken from a demo file provided by the IntelliJ SDK [3].



```java
containingMethod.accept(new JavaRecursiveElementVisitor() {
    @Override
    public void visitLocalVariable(PsiLocalVariable variable) {
        super.visitLocalVariable(variable);
        infoBuilder.append(variable.getName()).append("\n");
    }
});
```

Figure 12: Code snippet of visiting local variable

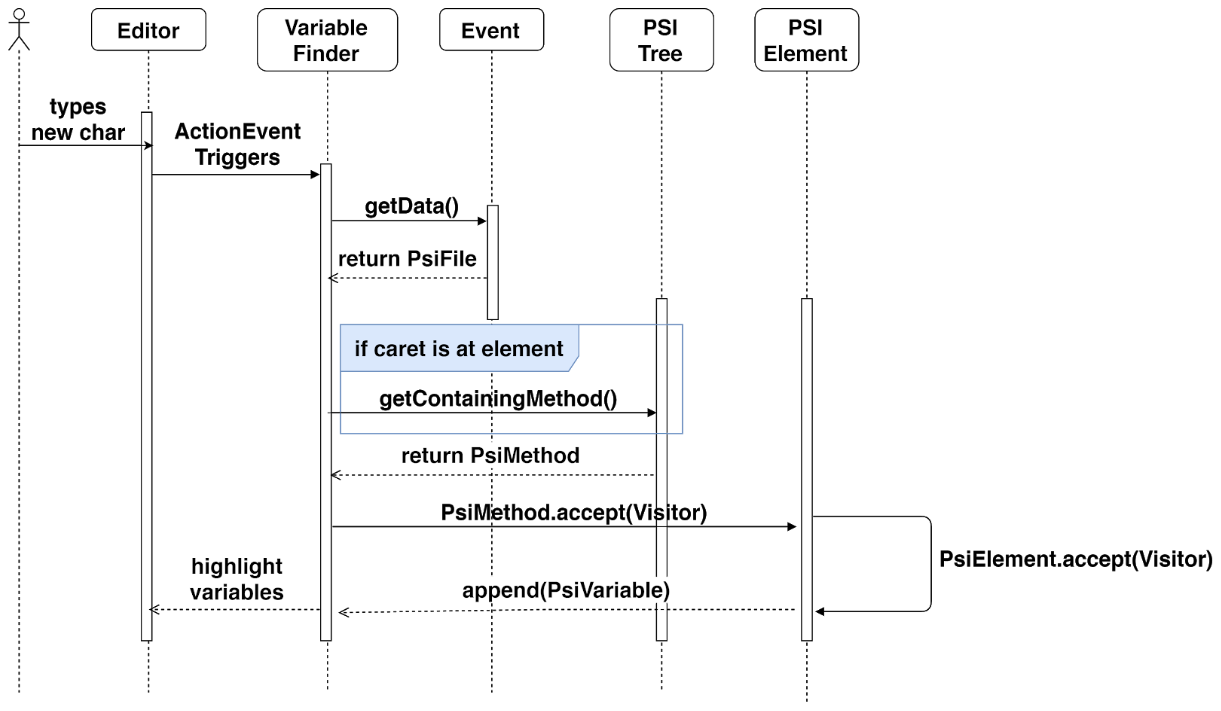


Figure 13: Sequence diagram of typing event

In the actual example, the developer creates a program to find all of the variables. An application of this may be to highlight all of the variables in the editor. The sequence diagram shows the control flow from the new character being typed and passed into the variable finder program. The variable finder accesses the data from the event, attempts to get the PsiMethod, and then search it for any local variables within it. The developer could then use this information to colour each of the variable names back in the editor. The full code can be found at [3] for more details.

5.0 Derivation Process

The main source that was used to extract the concrete architecture of the top-level systems and PSI was the provided LSEdit containment files and source codes. This provided the group with a substantial amount of information about the concrete architecture as it provided a diagram where components and their dependencies and hierarchies are visual. However, with too much information given, extraction of the PSI architecture became a problem which will be explained later.

Our first approach was to navigate through the diagram and extract the PSI architecture from there. But as we dug deep into the architecture, the diagram kept on getting messier. There were arrows with big arrowhead floating everywhere and components stacked together. Considering how massive the diagram was, it was almost impossible to derive the architecture from there. This led us to the idea of cleaning the diagram so that only components relevant to PSI are visible.

There were three LSEdit files that we fed to the software provided that makes the diagram:

1. `intelliJIDEA_File_Dependency.contain`
2. `intelliJIDEA_File_Dependency.raw.ta`
3. `intelliJIDEA_File_Dependency.con.ta`

With these files, we were able to read and see which components have a relation with the PSI but reading tens of thousands of lines seemed impossible. With the use of UNIX shell command ‘grep’, we were able to filter out components that we know for sure we do not need such as groovy, python, xml, etc. in item (1) which gave us a cleaner contain file to work around with. From there we read the new contain file and made a long list of keywords from the components that we need. Once we had a list of keywords and important folders, we started grouping those folders into subgroups similar to our conceptual architecture. From there, we had a few extra folders which did not fit into the other subsystems. So we knew we were at the point where we were dealing with divergences from the conceptual architecture. We grouped these folders into subsystems which we found were most representative of the leftover folders. Once we had the list of groupings, we added them to the original contain file and updated the raw and con (items (1) and (2) respectively) files to match. From here we could run the `createContainment` file to give us the updated landscape file which was run through LSEdit to give us the screenshots shown above.

6.0 Lessons Learned

Architecture extraction and analysis of big, complex software systems like IntelliJ IDEA can be quite challenging and requires nothing less of a herculean effort. And the tools that we currently have, which was developed some 20 years ago along with the primitive method taught in class, does not make it easier. While both jGrok and LSEdit are well-written and effectively do the job if you have unlimited time, they are not the most efficient way to analyze today’s software systems, especially ones where development is collaborative and where the code-base can scale exponentially. Fortunately, not everything went for naught as our frustrations led us to some key lessons.

1. People who study software architecture on a daily basis and document their findings are true academics.
2. Time and computing power are finite resources.
3. Software architecture is only meant to guide development, and not a be-all-end-all rule that must be adhered to.
4. Even the best programmers can be guilty of poor documentation.

5. Eyeballing file and folder hierarchies can be informative.

6. A manual approach to things may be more productive.

7. The software market for extraction and analysis of software architecture is due for an upgrade.

We also gained more appreciation for the ‘grep’ utility and shell scripts in this assignment. Both tools aided in navigating and filtering through the excess information within the contain and raw files.

7.0 Conclusion

To conclude, a conceptual architecture, especially for huge systems, does not always reflect the concrete architecture. In most cases, you’ll find unexpected dependencies between components that is not shown in the conceptual architecture. However, this does not automatically mean that the dependency is wrong. It’s just the reality of actual, practical implementation of software systems where requirements for a subsystem, a class, or a component can change through time. This should serve as a reminder of the importance of descriptive source-code comments, informative commits, and up-to-date documentation.

8.0 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

3. PSI Navigation Demo, IntelliJ-SDK-docs, JohnHake, JetBrains, https://github.com/JetBrains/intellij-sdk-docs/blob/master/code_samples/psi_demo/src/com/intellij/tutorials/psi/PsiNavigationDemoAction.java, Accessed 2019

4. Lamport’s Legion, “Conceptual Architecture IntelliJ IDEA”, Lamport’s Legion EECS 4314, 07-Oct-2019. [Online]. Available: https://jeremywinkler.github.io/pdfs/conceptual-arch.pdf