Design Documentation for Processing Language Server

Rationale

Processing Language Server focuses on creating a Language Server Protocol (LSP) implementation for Processing Programming Language. PDE is currently built using Java and using custom components of the Swing Framework, which is ~ deprecated. The long term goal of Processing is to replace this with a JS based IDE to bring in more contributors and to make building UIs simple. While planning on building such IDE, LSP is of significant importance for any language that the IDE relies on. Since Processing is the targeted Programming language, it's quite important to build a Language Server Protocol for the same. This shall act as a benchmark for all the crucial activities of the IDE such as auto-completion, go-to-definition, hover-insights and so on. LSP will also help in easy and seamless integration of the above functionalities in any editor such as Atom, VScode, etc.

Requirements

- Rich IDE functionalities for Processing from a remote server
- Basis for building new IDE for Processing
    - New IDE shall be built using JS - extending PLS.

Business Logic

Depends on the individual features that make up the whole system.

- **Features**
    - Auto-completion
    - Go-to definition
    - Hover for insights
    - Diagnostics
    - Finding References
    - Renaming / Refactoring

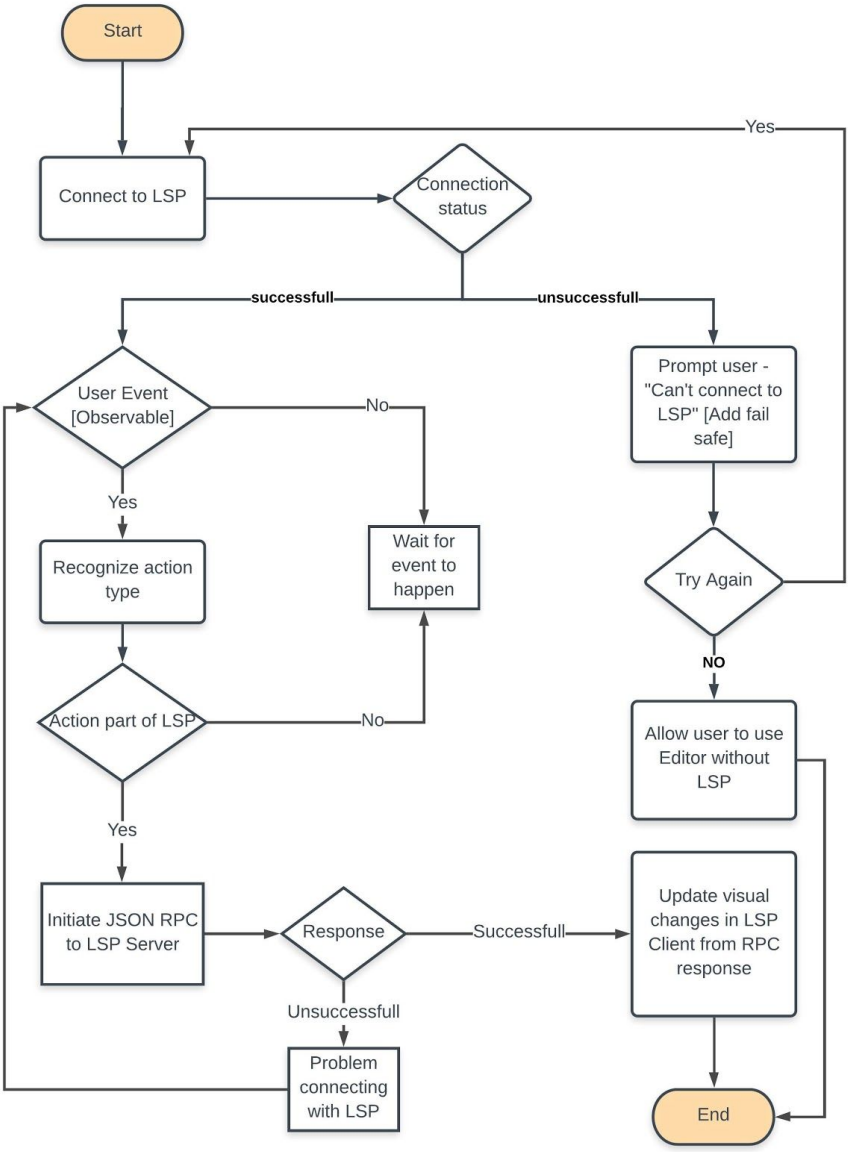
Architecture Diagram below gives an abstract overview of Business logic:

- **System Architecture**
    - The high level architecture of Language Support for Processing consists of two main classifications:
        - LSP Server
        - LSP Client
    - **LSP Server**
        - This is usually a remote server that has the implementation of all the above listed functionalities / features of Language Server.

    - **LSP Client**
        - This is a client side extension usually an extension of a generic code editor that consumes the Language Server and has the ability to perform all the functions based on the implementation found in the LSP Server.

LSP Client

- The Following Architecture diagram denotes the LSP Client:

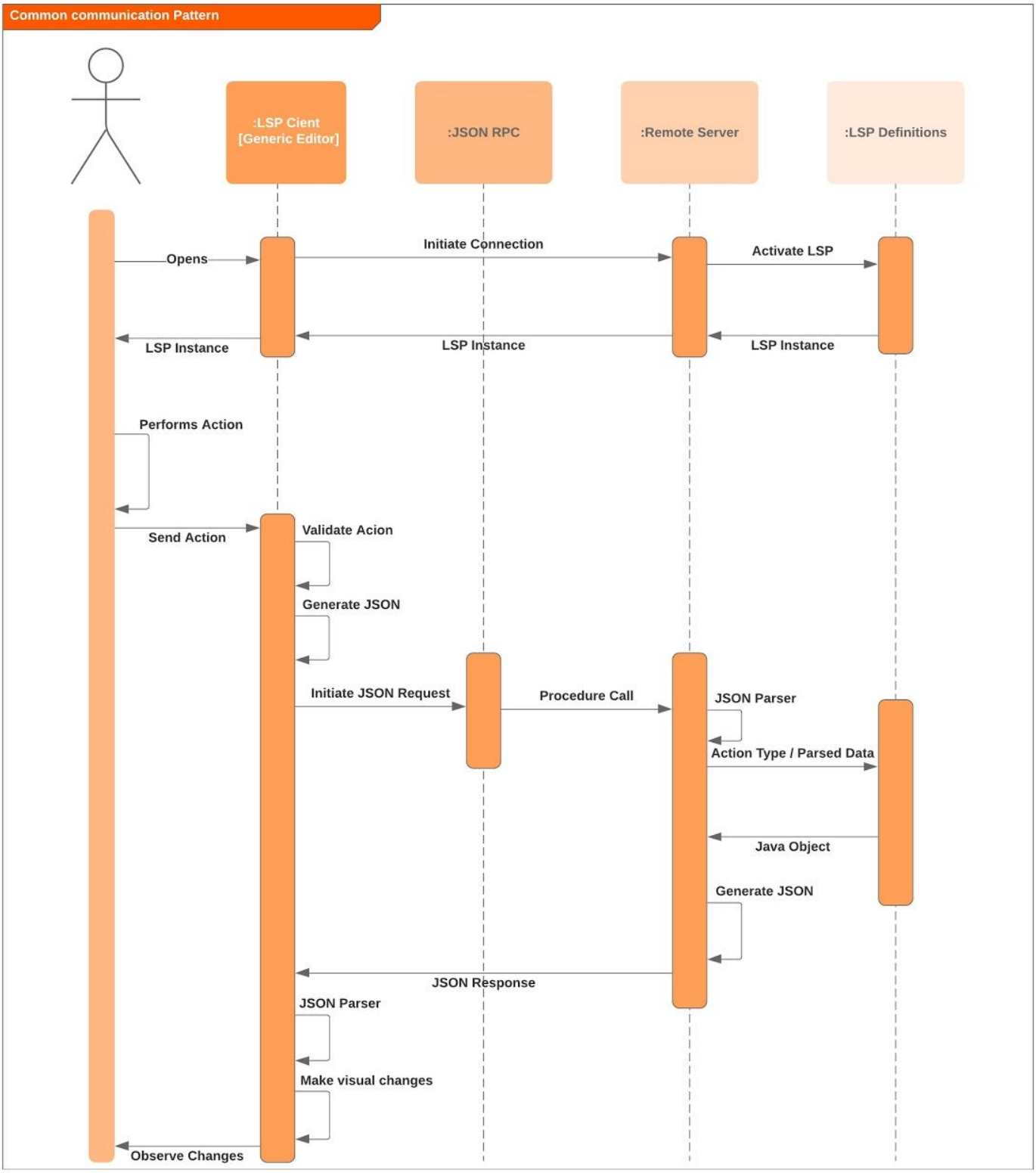


- **Remark**
  - **User Events**
    - This can be events such as Hover, Ctrl+click, Tab completion and so on.
    - Functionalities will be mapped with the type of user event performed.

  - **JSON RPC**
    - Stands for Javascript Object Notation Remote Procedure call.
    - This is the commonly adopted communication standards for LSP.

LSP Server

- The following Architecture diagram denotes the LSP Server:



Reason for Fragmentation

- The analysis tool can be implemented in any language, as long as it can communicate with the Language Client following the Language Server Protocol.
- As language analysis tools are often heavy on CPU and Memory usage, running them in separate process avoids performance cost.

Implementation Specific Details

Following are the implementation classes used in the server side:

- *Processing Language Server*
    - The class that initializes the connection to the client and maintains connection between Language Server and the Client until the end of the session.[from the server side]
    - Responsible for JSON Request / Response Processing - async.
- *Processing Text Document Service*
    - The class that is responsible for providing language specific details such as,
        - Auto-completion
        - Go-to definition
        - Hover for insights
        - Diagnostics
        - Finding References
        - Renaming / Refactoring
- *Processing Workspace Service*
    - The class that is responsible for maintaining the work-space data such as,
        - The number of files that are currently open [Tabs].
        - Current location of the cursor in a file when closed.
    - Maintaining this information is important because on reopen of the editor the workspace should be the same as when it's closed the previous time.

Following are the implementation classes used in the plug-in:

- *Processing Launcher Initializer*
    - Maintain connection between server and client from the client side.
- *Processing Event Identifier*
    - Identifies the type of event - Double click, Hover, Highlight Range, etc..
- *Processing RPC Parser*
    - Mapping event to Language Server Functionality and make request to the Language Server.

- ***Note :*** There is an interface available in LSP4J known as Launcher which initiates and maintains the connection between the server and the client on the client side .i.e., Plug-in, all the above classes depends on Launcher Interface.

References and Migration

- Currently PDE itself has come of the features that are listed above but they are available / coded with in the development environment. These functionalities from the PDE will be referred and possibly migrated to the LSP Server so that these features can be provided as a service to the LSP client as it extends from the server.

Request Response Structure

- Standard structure prescribed by LSP4J by Eclipse Foundation will be followed. Modifications not recommended or encouraged unless extremely essential if changes required should be decided initially, later change shall lead to reprogramming of the reception parser written inside LSP4J.

Interface

- The interfacing between the LSP client .i.e., the editor and the LSP server happens by means of Remote Procedure Calls .ie., JSON RPC. This happen through initial hand-shaking between the server and the client followed by a series of request response based on-demand which depends on the User events.
- Sockets are most preferred for communication since there are certain cases in LSP where Server initiates the request and sends it to client eg, request to updating the hashtable in the client side after performing diagnostics.

Client functions

- Extends LSP interface.
- Should be a generic editor
- Client receives object
    - Face object

Event identification on client server

- Ctrl+click / hover - should be distinguished - performed by client.
- Host client has inbuilt interfaces to distinguish these. Just extending the LSP shall do the trick. Each Operation is associated with the function callbacks in LSP (until what I've found by researching a few references).

JSON RPC encryption

- out of project scope can be considered for future development.
- can be serialized for networking purposes

Languages

- Typescript for LSP Server
- LSP Client depends on the generic editor
    - Typescript for VScode Client
- Shell scripts for file operations

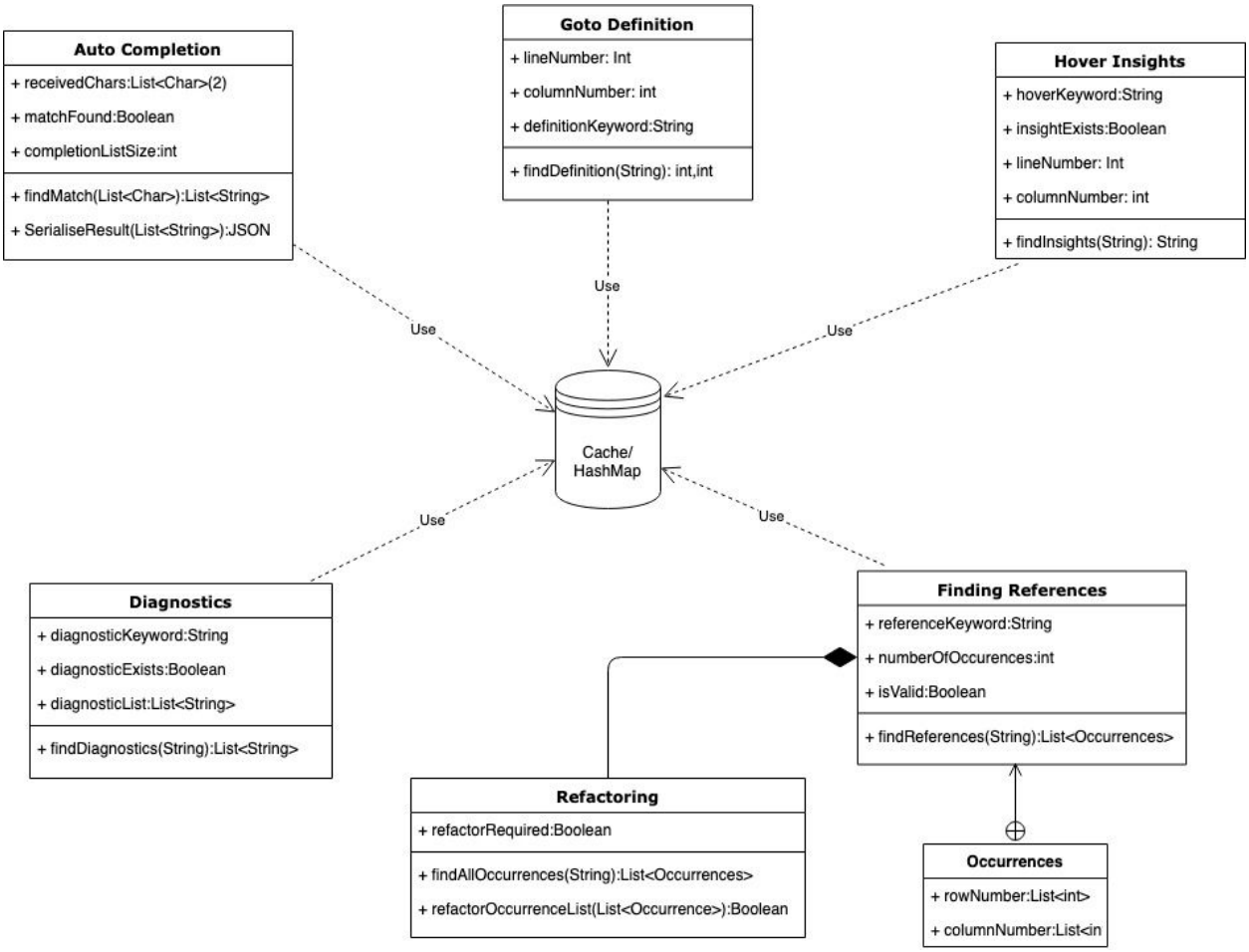
Alternatives

- Local running LSP
    - Multiple threads
- Remote server - real time
    - Web sockets

Frameworks

- Nodejs to provide functionalities for the server
- Npm / Maven to pull required resources from the web.

Components and their Relationship

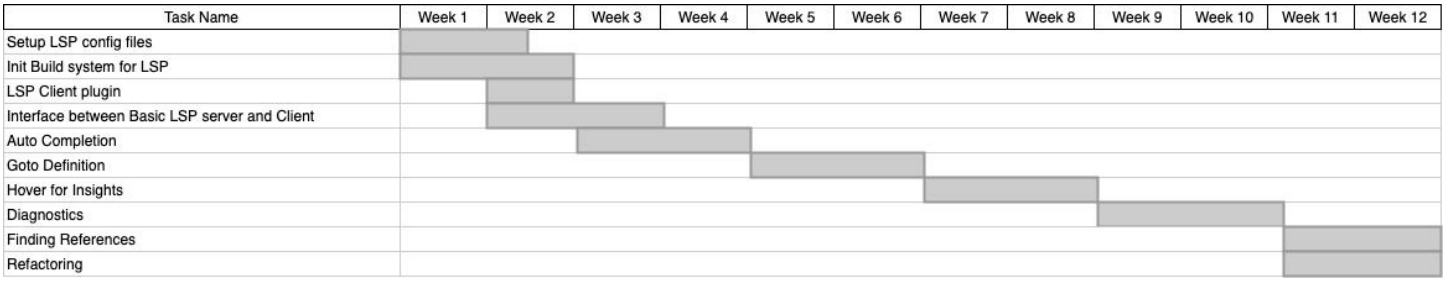


Risk

- On running on a remote server
    - Performance issues - latency of response
    - Where to host LSP if it were a remote server
    - Handle multiple concurrent requests to server
    - Optimisation Algorithms
    - Steps to reduce response time
    - Counter of clients that access concurrently
    - Concurrency upper bound
    - Scalability
- On running in threads
    - Deadlock when multiple clients
    - Server should give wrong respond to a wrong thread
    - Maintain identification using Thread ID

Development diagram

- Timeline diagram - week wise improvements
- Note - This denotes the anticipated results by end of each week but if some feature takes more time than anticipated the graph shall be altered to fit into the timeline without making changes to the deadline.



Testing / Debugging

- The communication between the client and the server can be managed using LSP Inspector [ ], which acts similar to that of a debugger that helps monitoring the data exchange as JSON.
- The code base debugging is done using standard TS debugger for LSP Server and Typescript debugger for LSP Client.
- Local machine
    - Thread setup - one running Server and the other running client.

Review

- Code clean-up by removing comments, TODOs and Server Logs.
- Automated Linting using Standard TS guidelines.
    - Lints will be checked for every commit but this is to ensure we've not left any code unmaintainable.

Release Management

Pre-release (Alpha / Beta)

- The pre-release will ship with an LSP Server and a VS Code extension containing the following features:
    - Auto-completion
    - Go-to definition
    - Hover for insights
    - Diagnostics
    - Finding References
    - Renaming / Refactoring
- The users can consume the Language Support functionalities of Processing by installing the VS Code extension which will be made publicly available under Processing Foundation in the VS code extensions Marketplace.
- Links and procedure for reporting issues will be provided in the extension index.

Stable Release (1.0)

- This release will be an increment over the pre-releases in terms of LSP Performance by employing cache to efficiently handle repeatedly used language functions.
- Fixing all potential bugs that were present in the pre-release.
- This release will replace the pre-release in the VS code extension Marketplace.

Abbreviations

- LSP - Language Server Protocol.
- PLS - Processing Language Server.