OML Alexandria

Team A1 - Pteam Ptolemy

Group Members:
Matthew Craig
Jeffrey Ma
Tamjid Khan
Jacob Linder
Jacques Gueye
Alexander Velikanov

Github: https://github.com/0x65-e/oml-alexandria

1. Final Design

Functional Requirements:

Since our project was proposed as a “proof-of-concept” for a Langium implementation of the OML language server, our functional requirements mainly consisted of language server features and a Sprotty diagram webview. In general, there was not much deviation from our original functional requirements, with one notable exception: the default implementation of code completion generated by Langium is not aware of the full or abbreviated IRI reference syntax of OML, and due to the complexity of the task in a limited time, we did not improve on the default behavior. Autocompletion is still partially implemented for keywords and locally defined IDs.

Due to the large size of our team, we were able to complete additional features beyond our original functional requirements. We added custom code hovering information not included in the original language server feature requirements, and we added an alternative method of visualization through PlantUML diagrams with a UML generator from OML source code.

Non-functional Requirements:

There were no notable differences to the non-functional requirements from our original design. Our non-functional requirements for scalability, reliability, availability and performance remained unchanged from our original project proposal, and our original selection of Langium as the language server framework that dictated the fulfillment of those non-functional requirements also remained unchanged.

Epics/Stories:

We added many user stories to expand on acceptance criteria for the validation requirements as we learned more about the syntax of OML. We also added user stories for the additional features mentioned above. We marked two of our existing stories as ‘wontfix’ - improving code completion beyond the Langium default (described above), and posting our extension to the Open VSX registry. We added tasks for development infrastructure, including our CI pipeline, unit testing, and automatic documentation generation. In line with our description of the functional requirements above, the only changes to our epics was to add specificity regarding acceptance criteria.

Architecture:

The final architecture landscape (monolith), the final architectural structure (layered), and the user-interface (Model View Controller) did not change from Part B. The only notable exception is that by adding the PlantUML generator, we added an additional component to the presentation layer of our application structure and an additional view to the user interface.

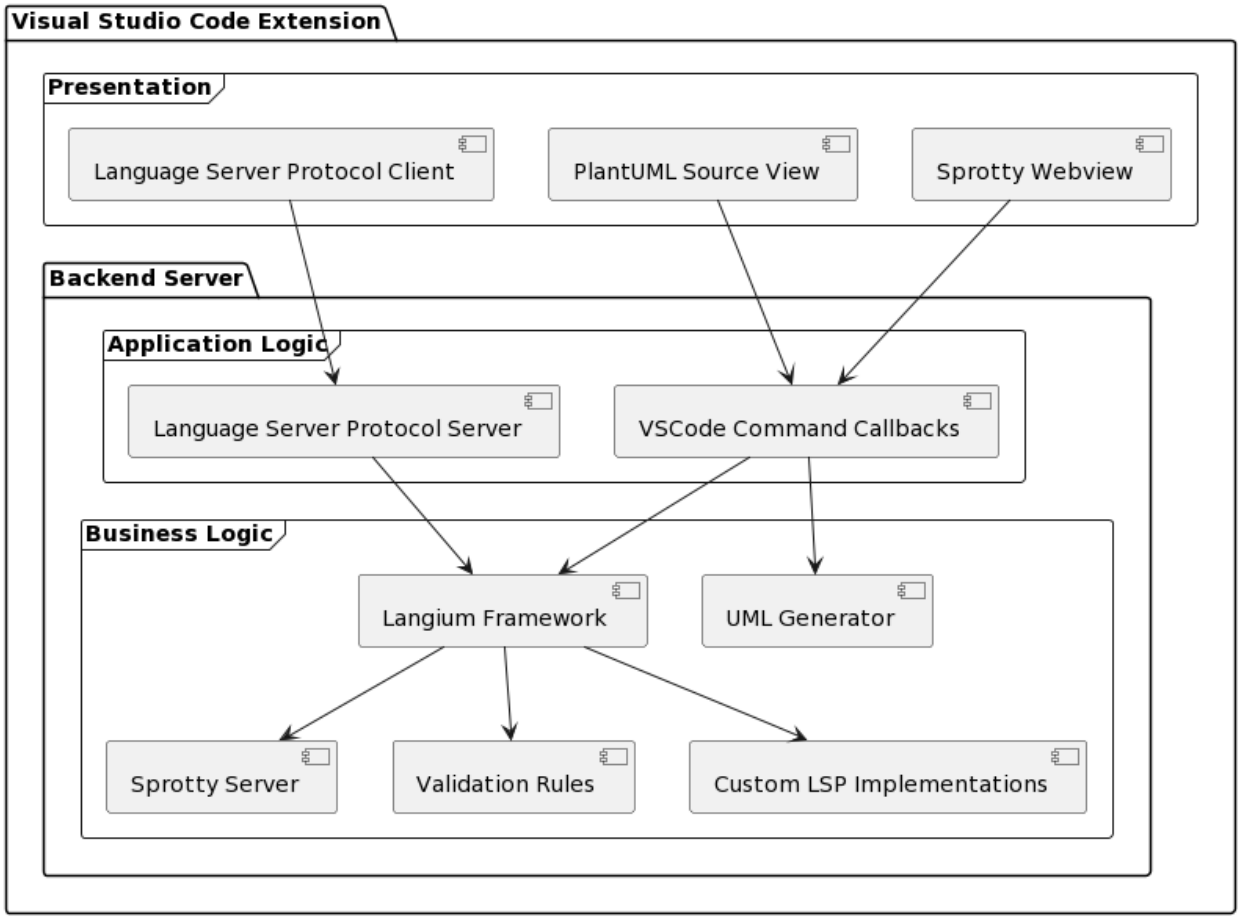

***Figure 1.1**: The final application landscape pattern is a monolith.*

Application Landscape. We chose a monolith for our application landscape architecture (above). The backend server runs in a separate process on the Node.js runtime, but the entire application is packaged as one executable (extension). A monolith is the fastest architecture to develop, and is the easiest architecture to integrate a tightly coupled frontend and backend. Since a language server instance must be specific to a single workspace in order to provide context-dependent features like find-references or go-to-definition, a monolith does not penalize our scalability. Finally, a monolithic application ensures reliability and availability since the entire extension is available and runs locally, so we don't have to worry about communication between machines or complex coordination between processes.

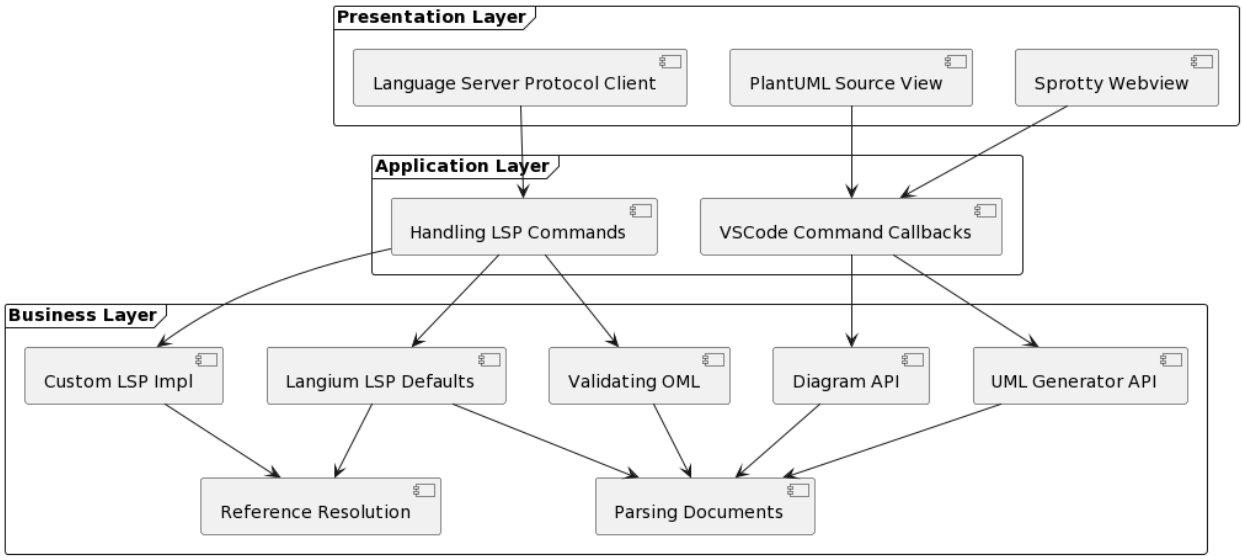

***Figure 1.2**: The final application structure pattern is a layered architecture.*

Application Structure. Our application structure is layered, but with only three layers: presentation, application, and business. The Sprotty webview, PlantUML source view, and the LSP client provided to VSCode compose the presentation layer. These user interfaces interact with the LSP server and the command callbacks registered with VSCode to invoke actions on the backend and request input. The business layer consists of the language server features provided, including both Langium defaults and our own implementations; the functions that those LSP features rely on (which includes parsing documents and resolving cross-file references); and the APIs for generating Sprotty diagrams and PlantUML code. Since our application doesn't store any data, we don't have a persistence or database layer.

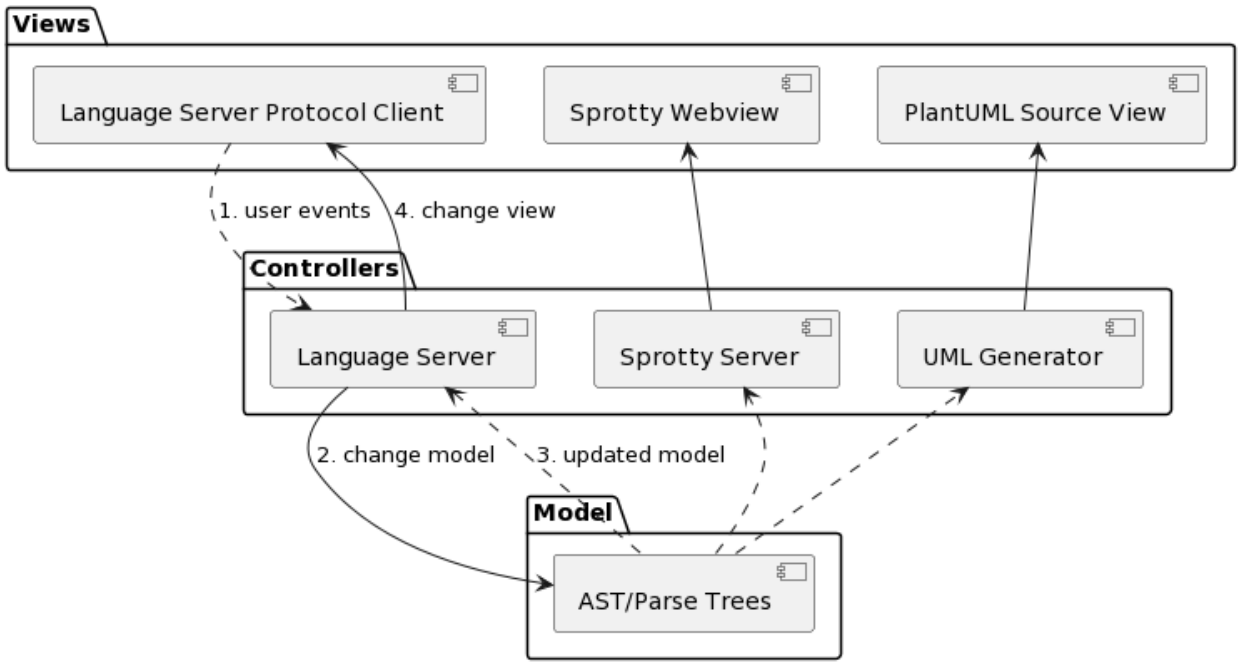

***Figure 1.3**: The final user interface pattern is Model-View-Controller (above)*

User Interface. Our application implements three different methods of displaying information to the user, which we have termed as our views. These are: LSP features (e.g. syntax highlighting, hover text, reference locations), a Sprotty diagram webview, and PlantUML source code. Each view is coupled to its own controller that changes the view. Since there is currently no way to change the underlying files through the two visualizations, we have left off those connections in the diagram. All three controllers interact with the model, which is the trees parsed from the underlying OML files. Updated models are made available to each controller to update their respective views.

Software Design:

Our application consists of two components. The first (and simplest) is the language client, which is a VS Code extension responsible for starting the language server and for displaying visualizations to the user. We will refer to this component as the frontend, since it routes requests for language features to the second component. The second component, which we will refer to as the backend, is the language server, which runs in a separate process on the Node.js runtime. The backend has three major responsibilities: first, to parse documents written in OML grammar and construct both an abstract syntax tree (AST) and a parse tree; second, to handle requests from the frontend that implement the Language Server Protocol; and third, to traverse the parse tree and construct a visualization when requested by the frontend. Since the majority of our application development is focused on the backend server, we will focus on the three major design patterns that the backend server implements.

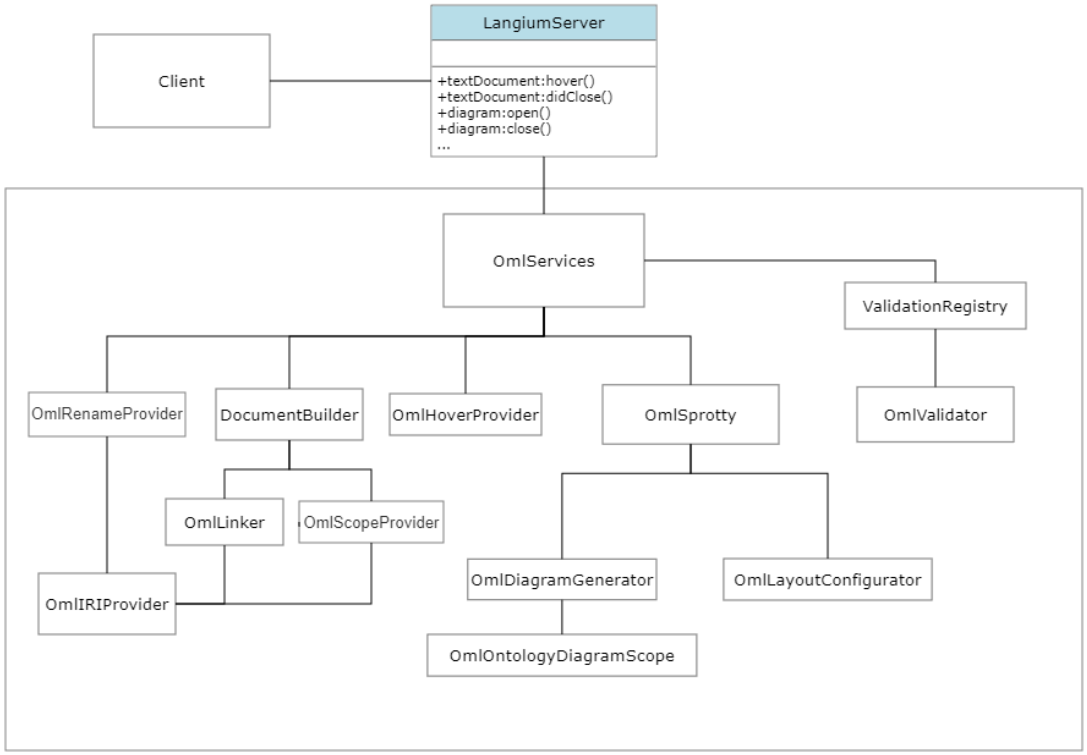

***Figure 1.4**: Final class diagram of design using the facade design pattern. This class diagram is of the language server backend.*

Facade Pattern. A simplified diagram of the language server backend design can be seen above. The first design pattern that the server implements is the Facade pattern. The server provides a simplified interface for the client, with methods to allow the client to update the server on opened documents, changed documents, and request a language capability that the server supports. The server is responsible for dispatching the request to the correct service to handle the request. In the diagram above, these services are connected to the OmlServices class and include DocumentBuilder, OmlRenameProvider, and OmlHoverProvider, as well as many others not pictured. Some of these services are generated by Langium, and some were written by us to extend the Langium default implementations. This pattern decouples the client from the many implementing classes contained in the server and permits simplified communication over an interface, the Language Server Protocol.

Visitor Pattern. The Visitor pattern is the second major design pattern that the backend implements. The Visitor pattern is used in the construction and traversal of the syntax and parse trees. Construction occurs any time an OML document is created or changed. Traversal occurs in many use cases, including validating the parse tree for semantic violations and traversing the syntax tree to create a visualization for the user. In these cases, it is very useful to have a Visitor pattern that can perform the appropriate action on each type of node, like checking against the correct validation patterns or constructing a UI element to represent the node visually. We omit a diagram here because the concept is simple, and there are a lot of Visitor classes to display in one diagram.

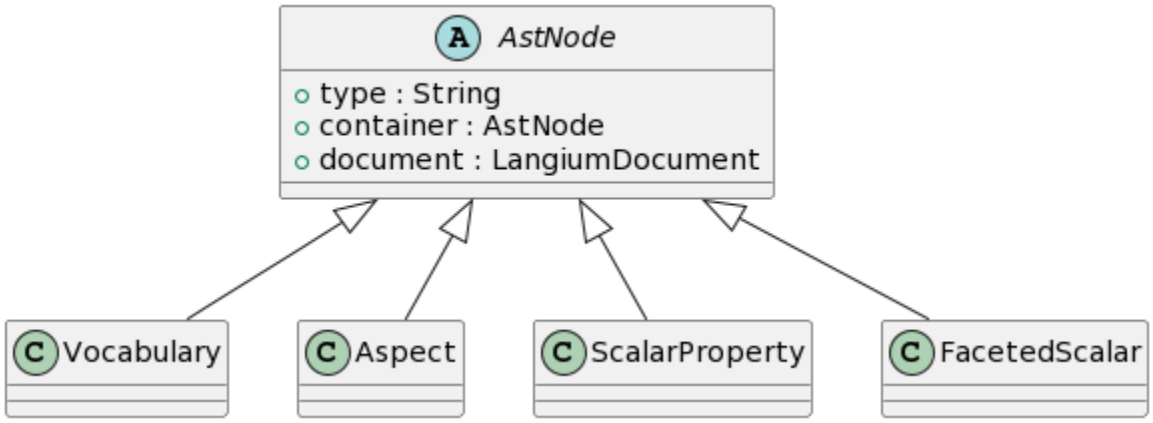

***Figure 1.5**: The parse tree follows the composite pattern.*

Composite Pattern. The third design pattern is the composite pattern, which is used in several places. The first is in the abstract syntax tree and parse tree itself (above), which uses the AstNode class. A tree or subtree is treated the same, regardless of its depth. The second is in combination with the Visitor pattern for validating the parse tree. Multiple validation rules can be composed together to apply several validation rules to the same node type using the ValidationChecks type. This simplifies the implementation of the validation visitors.

2. API Documentation

Our extension is written entirely in TypeScript, and we used TypeDoc, a popular documentation generator for TypeScript, to automatically generate our API documentation. Our generated documentation is available in the “docs” folder of our repository, and is hosted via GitHub Pages.

Live Docs: https://0x65-e.github.io/oml-alexandria/
Source: https://github.com/0x65-e/oml-alexandria/tree/main/docs

3. Test Cases

Our test strategy was to test the major components of our language server, as this was where a vast majority of our efforts went. Key components like our syntax tree and validators needed to work, therefore we had to create tests for them that would run before any push. The framework we used for testing was Vitest. We used this framework to check major functionalities and important methods, all of which can be run via ‘npx vitest’. We created multiple test files with different test cases, like the ones shown below.

Example Test Case 1
Link: https://github.com/0x65-e/oml-alexandria/blob/main/test/nodelocator.test.ts#L20-L33

A major part of our project is being able to traverse the syntax tree so that all nodes are clearly present and able to be reached by some path. Issues with this can mean that the AST was generated incorrectly and can render the extension broken.

The first test case checks to make sure that given the node, a path exists to that node. It takes in a golden test file, picks a specific node, then uses Langium’s test method to create a path to the node. For the test to pass, we expect the method to return the correct path.

The second test case does the opposite of the previous one: it checks to make sure that given the path, a node exists on the path. It picks a specific node, then uses Langium’s test method to find the node on that path. For the test to pass, we expect the method to return the correct node.

Example Test Case 2
Link: https://github.com/0x65-e/oml-alexandria/blob/main/test/cross-refs.test.ts#L26-L49

Another important feature of the project is the ability for the extension to resolve cross-references between multiple files/vocabularies. If one vocabulary extends another, the extension must be able to find all symbol references to the latter vocabulary.

This test case checks to make sure that cross referencing still works with our language server. It compares two specifically created OML files (one declarative and one referencing) and uses Langium’s test method to find all references between them. For the test to pass, we expect

the method to return the correct number of references (1), whereas a failure could possibly imply a problem with our syntax tree.

Example Test Case 3

Link: https://github.com/0x65-e/oml-alexandria/blob/main/test/validator.test.ts#L77-L91

The ability of our extension to validate OML files for semantic and syntactic consistency is a major feature. Different members have multiple rules, so we created a validator. A major portion of our code went into this section, so it was important to test these validator methods to make sure they work as intended.

This specific test is for one of the validator checks: 'checkEnumScalarNoDuplications'. The OML object 'EnumeratedScalar' contains multiple unique literals; this validation checks that the 'EnumeratedScalar' does not contain a duplicate literal. The first test case just checks that a normal EnumeratedScalar doesn't return an error. It takes a proper EnumeratedScalar, then calls the 'checkEnumScalarNoDuplications' with it as the parameters. To pass, we expect it to return no errors. The second test case then checks that an EnumeratedScalar with a duplicate literal does return an error by doing the same call like in the first case. Failures for either of these cases means that the 'checkEnumScalarNoDuplications' method was changed improperly.

In general, the test cases we used cover the language server fairly well. In terms of coverage, the one area that could have been improved on is the validation testing. Given that our validator contains dozens of validation check methods, the way to achieve full coverage would be to make unit tests for each of them. However, we found it satisfactory for the project to just make unit tests a select few of these methods instead, and focus our development efforts on other features. We tested the validation rules manually, by creating an improper OML file and looking for expected warnings and errors.

4. CI/CD

The first step in code integration was manual code review and approval from another developer on every pull request. Generally, the backend team would approve each other's pull requests and the frontend team would do the same. Pull requests were linked to GitHub issues (tasks and stories) so the reviewer could verify that the pull request met functional requirements and suggest additional improvements.

In addition to manual review, we built an automated CI/CD pipeline using GitHub Actions that ran on each push and pull request within the repository. We built on Ubuntu 22.04 for a consistent build environment, and the pipeline consisted of building the repository, running a linter over the code, and finally running the unit tests. This ensured that our changes did not disrupt the build process or the current functionalities tested by unit tests, as well as ensuring the TypeScript was properly and consistently formatted using ESLint. The CI/CD pipeline was required to pass in order for pull requests to be merged into the main branch.

Since we don't have a hosted deployment (the user installs the extension themselves), we did not require a separate CD pipeline. Publishing new versions of our extension to the VS Code Extension Marketplace was handled manually.

5. User Manual

GitHub Wiki: https://github.com/0x65-e/oml-alexandria/wiki

6. Scrum Summary

We split our work into three sprints of two weeks each. We focused our work on four epics: (1) to run the extension in browser-based editors; (2) to provide basic language features like Go to Definition, Find All References, and Autocomplete; (3) to visualize OML files using diagram web views; and (4) to provide strong semantic validation for OML files. We were able to complete all of our planned features, and even expanded on our original proposal by adding additional visualization features.

Sprint 1

Our first sprint revolved around creating our application model and code generation configuration for Langium. This consisted of translating the OML language syntax from the Xtext source of truth into the Langium grammar. We encountered some difficulty achieving the same expression in both formats, which did set us behind our original plan. We spent a significant amount of effort resolving errors in the generated Langium grammar that prevented us from generating a working prototype. Our slow start bottlenecked progress for our frontend team, which was unable to begin work until our code generation was completed.

In this sprint, we completed 4 user stories and 2 tasks (note that the tasks are backdated to accurately reflect work completed during this sprint, per discussion with Apoorv).

Sprint 1 Contributions

Jeffrey

Epic #1 (browser support) - I worked through the Langium tutorial to understand how we could use Langium to implement our language server on the Node.js runtime.

Epic #4 (OML validation) - I read through the OML specification to understand the semantics of OML.

Task #53 (run xtext2langium) - I ran the xtext2langium fragment on the OML repository. I then verified that the generator had properly converted the grammar rules by inspecting each rule and comparing it to a given BNF grammar.

Task #52 (run langium generate) - I created the generated parser/AST types using Langium generate. I also noted that Langium did not properly propagate types to superclasses when generating, manually doing it to fix the generated ast.ts file.

Issues #6, #8, #9, #12, #13 (single file language support) - I fixed roughly 200 errors that arose from erroneous conversion from the xtext2langium generator, preventing Langium from generating the parser. This includes changing rules to infer union typing instead of the improper interface types (which are not supported), modifying class hierarchy, changing unordered groups to Langium supported syntax, and changing rules to properly create objects. This

enabled all basic language feature functionality. I verified that language features with a single file work by writing a cli function to parse and validate with Langium.

Matthew
Epic #1 (browser support) - I read through the Langium tutorial.
Epic #4 (OML validation) - I read through the OML specification to understand the semantics of OML.
Issues #6, #8, #9, Issue #54 (single file language support) - I resolved some errors with terminal production rules in the grammar that were causing the parser to fail and wrote some functions to help debug the parsed AST.

Alec
Epic #1 (browser support) - Spent a meaningful amount of time researching how Langium vs-code extensions work and how Langium code generation works. Attempted to help debug the grammar issues (albeit without much luck)

Sprint 2

Our second sprint contained the majority of our backend work for language feature support, and the initial exploration stages of our frontend development. We solved issues related to cross-file reference resolution, enabling the use of our extension for a whole OML project and effectively completing all of our original functional requirements for language server protocol support. We also reduced tech debt accumulated during our first sprint to get Langium working initially, and began planning validation acceptance criteria for our next sprint.

Jeffrey
Task #56 (fix relation references) - I fixed relation references by modifying Langium local scope to also include forward and reverse relations for Relation entities.
Issue #10, Task #57 (cross-file references and find-all-references) - I fixed cross-references by modifying the exports to global scope to use Full IRI syntax. Additionally, I modified the default Langium linking to convert abbreviated IRI syntax to Full IRI at linking time. This enabled language features such as goto definition and find-all-references.

Matthew
Task #55 (remove union types) - I removed most of the union types inferred by Langium that we used to get the grammar working, but which were cluttering up the generated code (tech debt).
Issue #7 (syntax highlighting) - I wrote a TextMate language grammar to add syntax highlighting for OML.
Issue #9 (code completion) - After investigating Langium and OML, I decided that we could not provide more specific code completion and closed this issue.

Jacob
Epic #4 (OML validation) - I read through the OML specification to understand the semantics of OML. This enabled me to work with the project's backend. I also searched through the OML

documentation to create a thorough list of validation tests across the OML grammar that we could implement.

Sprint 3

Our final sprint focused on frontend visualization, validation, and our testing/deployment infrastructure. We integrated the oml-sprotty webview and replicated the backend functionality of the existing Java Sprotty server. Since we were no longer bottlenecked and could utilize all of our developer resources, we expanded on our original project proposal by including additional language features like hover information and developing a PlantUML generation tool.

Jeffrey

Issue #11 (renaming) - I modified the Langium rename provider to properly handle ID, abbreviated IRI and Full IRI syntax (assuming that you want to rename the target node).
Epic #3 (hover) - I modified the Langium hover provider to give OML specific information about the referenced construct, including annotations, specializations, and other information.

Matthew

Issues #19, #21, #26 (specialization validation) - I wrote validation rules to ensure that specializations were correct (correct subtypes/supertypes, no duplicates, correct number).
Issue #18 (duplicate names) - I wrote a validation rule to ensure that all IDs were globally unique within an ontology.
Issues #14, #15, #16 (visualization) - I implemented most of the backend of the Sprotty server to generate the ontology diagram view, including computing the scope of displayed elements, generating nodes and labels, and writing functions to traverse the AST to find elements.
Task #64 (release v0.0.1) - I added documentation to the README and created a logo for the extension.

Jacob

Issues #22, #25, #31, #32, #37, #39, #30 (validation) - Created a set of validation tests, primarily for the vocabulary parts of OML; validation covered things such as semantic consistency for rules, axioms, properties, concepts, and relations
Task #47 (CI builds) - Wrote the script that allowed for automated building, linting, and unit testing for every commit push

Alec

Issues #14, #15, #16 (visualization) - Integrated the updated oml-sprotty submodule, set up launching the webview, did much of the research work on how the backend Sprotty server worked, contributed the axiom edges and other portions of the sprotty visualization backend
Task #64 (release v0.0.1) - Configured build process including building oml-sprotty submodule, moving webview file into correct place, and other bundling optimizations
Issue #5 (VSCode Marketplace) - Set up our publisher account and released the package onto marketplace

Epic #2 - Modified the oml-sprotty library to use the latest version of Typescript, Sprotty libraries, Webpack, and other dependencies. Consequently this required updating various configs, imports, and other miscellaneous code changes.

Tom
Issues #14, #15, #16 (visualization) - I wrote the Typescript scripts that convert .oml files to .plantuml files that can either be transferred to an online plantUML visualizer or be viewed within VS Code using a plantUML extension. I also added the Generate UML command that can be run by right-clicking any .oml file.
Epic #2 (visualization) - UML representation of .oml files

Jacques
Issue #14 (visualization) - I helped finish up our backend to generate the ontology diagram view, mainly implementing computing the scope of elements.
Task #50 (unit testing) - I created all the test cases for our language server using vitest, including cross-referencing and validation checks.