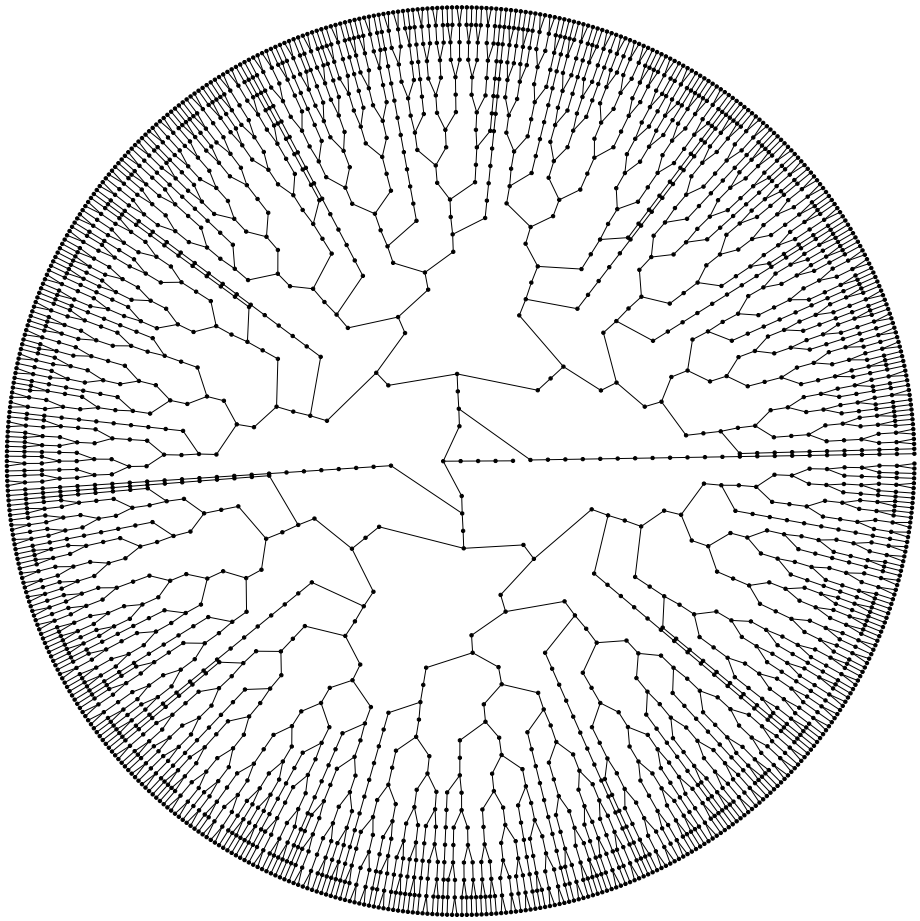
Portable Editor Services

Master's Thesis



Daniël Pelsmaeker

Portable Editor Services

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Daniël Pelsmaeker
born in Enschede, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Hailstone sequence of the Collatz conjecture, shown for positive integers up to 28 steps from 1, laid out in a circle around 8.

Portable Editor Services

Author: Daniël Pelsmaeker
Student id: 1367838
Email: d.a.a.pelsmaeker@student.tudelft.nl

Abstract

Implementing the syntax and semantics of a new (domain-specific) programming language is a lot of work, which is worsened by the additional work needed to add support for the language to an editor such as Eclipse or VS Code. Lack of such support can impede language usability and adoption, as developers prefer different editors. However, supporting m editors for n languages requires $m \times n$ implementations to be built and maintained, which is known as the *IDE portability problem*. Portable editor services aim to reduce this to $m + n$ implementations, which leads to the main question of this thesis: how can we make the editor services of languages portable across editors?

Language definitions made in the Spoofax language workbench can automatically expose their editor services in any editor that Spoofax supports. Therefore, we evaluate the portability of Spoofax Core, the editor-agnostic core of Spoofax, through an implementation of the workbench in the IntelliJ editor, and discuss the challenges we faced.

To get portability for editor services of languages in general, we first investigate how editor services can be added to the most popular editors, and explore their features, documentation, and application programming interfaces. From this, we derive a platform-agnostic model for portable editor services: AESI, the Adaptable Editor Services Interface. AESI describes the maximum set of common editor service features supported by the editors we investigated, while at the same time imposing minimal requirements upon any implementation of these editor services.

We evaluate AESI by providing two language implementations, and adapting AESI to Eclipse, IntelliJ, and VS Code. Finally, we compare AESI with other solutions to the IDE portability problem, such as Language Server Protocol and Monto.

Thesis Committee:

Chair:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. S.T. Erdweg, Faculty EEMCS, TU Delft
Committee Member:	Dr. ir. A.R. Bidarra, Faculty EEMCS, TU Delft
University Supervisor:	Ir. G.D.P. Konat, Faculty EEMCS, TU Delft

Preface

I would like to thank my supervisors Gabriël Konat and Eelco Visser for the opportunity to work on this project, and their time, patience, insights, and helpful feedback. I have learned a lot about doing research in general, and the field of programming language research in particular, and will apply this knowledge throughout the rest of my career. I am also grateful to Danny Groenewegen and Sven Keidel for their help, and to the members of the Programming Languages group for the interesting discussions, and the motivating and productive working environment. Last but not least, a big thanks to my family for their support during my time as a student.

Daniël Pelsmaeker
Delft, the Netherlands
April 25, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming languages are formal languages that provide software developers with a readable way to express the instructions for a computer to execute. There exist general-purpose languages such as Java, Rust, and JavaScript which can be used in a wide variety of application domains. On the other hand, domain-specific languages (DSLs) have constructs for a specific intended application domain, such as HTML for webpages, SQL for querying databases, and Verilog for describing integrated circuits. Hundreds of programming languages already exist, and new languages are continuously being created because of new application domains, advances in computing, increases in computing power, and new insights into programming language paradigms.

Most developers write their programs in a textual code editor. This can range from a simple text editor to a fully-featured integrated development environment (IDE). Code editors provide *editor services* that aid the user while reading and writing code. *Syntax coloring* is the most common editor service provided by even the simplest of code editors, where the various words and constructs in the code are colored according to their syntactic meaning to improve the readability of the code. A *debugger* is an example of an editor service that tends to be found only in full-featured IDEs, and allows the user to step through a running program and inspect the state of the program at each point in the execution. Most code editors can load plugins that add editor services for languages that are not supported out-of-the-box.

While a new programming language may share concepts with existing programming languages, it will often share none of its implementation, and consequently, the new language will have no support in any code editor. Next to implementing the language from scratch, another way to mitigate this is to use a *language workbench* (Fowler 2005), such as Xtext (Eysholdt and Behrens 2010) and Spoofax (Kats and Visser 2010). A language workbench provides tools for defining the language, and uses the language definition to create a language implementation with editor services.

However the language is implemented, the language developer runs the risk of tying the implementation too much to one particular editor, making it hard if not impossible to ever support other editors. Since users have vastly different preferences when it comes to editors, there is no single editor that would satisfy all users. Therefore, support for multiple editors could improve the adoption of the language. Even if it is possible to adapt a language implementation to various editors, it may not be feasible to write a plugin for every editor. Writing each of these editor plugins for a language is a lot of work, and each of these plugins must be maintained in the face of changes to both the editor and the language itself. This effort has to be repeated for every new programming language, and for every editor that needs support. This has been named the *IDE portability problem* by Keidel, Pfeiffer, and Erdweg (2016): supporting m languages in n code editors requires $m \times n$ editor plugins, one for each combination of a language and editor. This leads us to the main question of this thesis: *how can we achieve portable editor services?*

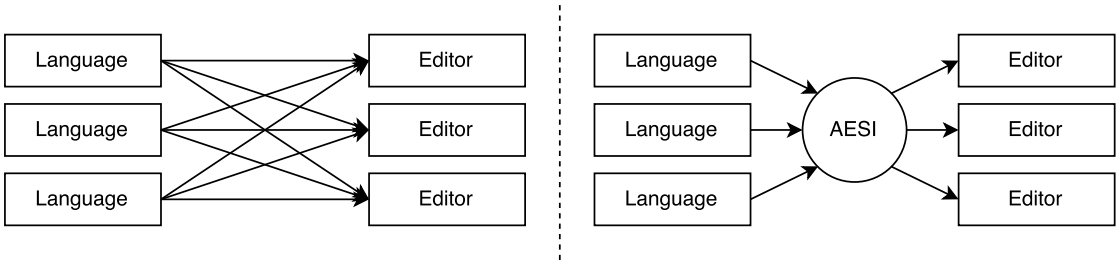


Figure 1.1: The IDE portability problem and its solution in AESI. Supporting m languages in n editors normally requires $m \times n$ implementations, shown on the left. A solution to the portability problem reduces this to $m + n$, shown on the right.

We first analyze the portability of the Spoofax language workbench by implementing its core in a second editor. Then we explore how a language is implemented on two platforms: Xtext, and IntelliJ. As editors impose different requirements and restrictions upon new editor service implementations, we examine those next. Finally, we use what we learned to derive a model for portable editor services and evaluate it.

As a first step, we studied the portability of the Spoofax language workbench. For a language definition in Spoofax, the workbench can provide the syntax coloring, code completion, analysis, and reference resolution. Spoofax traditionally used the Java-based Eclipse IDE as its sole platform of choice, but more recently the core of the Spoofax workbench has been rewritten to be independent from Eclipse. This new platform-agnostic core of Spoofax is aptly known as Spoofax Core, and porting it to another platform should give insight into the flexibility and portability of Spoofax Core and the complexities of supporting multiple code editors.

We ported Spoofax Core to the popular IntelliJ platform, which is a fully-featured Java-based IDE by JetBrains. While this proved that Spoofax Core is portable, it revealed a maintenance burden. The implementation needs to be maintained next to the existing Eclipse implementation, in the face of changes introduced by new releases of both editors and the evolution of Spoofax Core. For example, Spoofax Core is expected to get a big overhaul when a new system for incremental editor services is integrated into it.

So we asked ourselves: can we generalize the editor services such that even big changes to their implementation require little to no change in their ports? Perhaps we can generalize the editor services such that they are not tied to Spoofax Core at all, and leave the Spoofax-specific details out. This would open the door for any language to implement these editor services, not just today's Spoofax Core. After all, we can only guess at what the future would bring to Spoofax.

There are existing solutions to the IDE portability problem that define an interface for editor services: Monto, and Language Server Protocol (LSP). Monto, proposed by Keidel, Pfeiffer, and Erdweg (2016), is a language- and editor-agnostic intermediate representation and service system. The service system manages language services with whom the editor services communicates using messages adhering to the Monto intermediate representation. LSP (Microsoft 2018a) is the definition of only an intermediate representation to be used between a language server and an editor's services. We discuss both these technologies, and compare them to our solution, in Chapter 6.

As there are many ways in which editor services can be implemented by a language, we first wanted to get some experience building a language. For one simple language we wrote two implementations: one for the IntelliJ IDE and one for the Xtext workbench. To be able to properly generalize the editor services, we then gathered the requirements imposed on implementations of editor services in various editors.

Based on what we learned, this thesis proposes a solution to the editor services portability problem: the Adaptable Editor Services Interface (AESI), a model for portable editor services that can be implemented for any language and adapted to many code editors. Abstracting the editor services this way is a solution to the aforementioned problems of portability and maintenance, as shown in Figure 1.1.

This thesis continues as follows: in Chapter 2 we discuss the challenges of porting Spoofax Core to IntelliJ and our insights. To get more familiar with normal language implementations, we explore the implementation of a language in IntelliJ and Xtext in Chapter 3. We then take a look at the most popular code editors in Chapter 4 to find out what requirements they impose on their editor services. Chapter 5 describes AESI, our model for an adaptable editor services interface, which we derived from what we learned. We evaluate AESI by providing two implementations and three editor adapters for four editor services. In Chapter 6 we show how our model differs from two other intermediate language services: Monto and LSP, and end with the conclusion in Chapter 7.

Chapter 2

Porting Spoofax Core to IntelliJ

Language workbenches are tools that assist in the development of programming languages, supporting the efficient definition and reuse of languages by providing a language engineering framework and editor services for these languages. They are made possible by the similar analyses and editor services that programming languages require, even when their syntax and semantics differ.

Spoofax is a language workbench developed at the TU Delft. Historically, Spoofax has been inextricably tied to the Eclipse integrated development environment (IDE) through its use of the Eclipse IDE Meta-tooling Platform (IMP) (Charles, Fuhrer, and Jr. 2007), which provides editor services such as syntax coloring and reference resolution for Eclipse.

Recently, Spoofax has been rewritten to be independent of Eclipse, to pave the way for Spoofax to be used in other editors. This produced a portable core, known as Spoofax Core, a platform-agnostic library which allows any editor, not just Eclipse, to interface with the Spoofax services. Implementations of Spoofax Core were written to support the command-line and Eclipse (replacing IMP), with the intent that more editors could be supported in the future.

In this chapter we study portable editor services in the context of language workbenches, and Spoofax in particular, by adapting Spoofax Core to a different IDE: JetBrains' IntelliJ. This will allow us to gain knowledge in implementing portable editor services, while evaluating whether Spoofax Core is truly platform agnostic. Additionally, this should allow current and future Spoofax programming languages to be both developed and used in IntelliJ as well as Eclipse.

We chose IntelliJ as it is a popular IDE for the development of programs in Java and Java-compatible languages. Google switched from Eclipse to IntelliJ as its base for Android Studio (Toporov 2013), and the Xtext language workbench started adding support for IntelliJ as well (Oehme 2015). IntelliJ is written in Java, which makes it a good target for porting Spoofax Core, being Java-based as well. However, internally IntelliJ is very different from Eclipse, as IntelliJ provides more language support infrastructure. This makes it easier to add IntelliJ-specific language support from scratch, but makes it harder to integrate an existing language's services.

The chapter continuous as follows: in Section 2.1 we discuss the details and challenges of porting Spoofax Core to IntelliJ, and we end with a conclusion in Section 2.2.

2.1 Challenges and Observations

This section discusses the challenges and interesting implementation details we encountered while porting Spoofax Core to IntelliJ.

2.1.1 Dynamic Language Loading

One feature that sets Spoofax apart from some other workbenches is the ability to load languages dynamically, which enables the language developer to perform a quick edit-compile-evaluate cycle without having to start a new editor instance to see the changes. Similarly, the developer can load and unload meta-languages used for language development on an as-needed basis without restarting the IDE.

Spoofax Core provides its `LanguageDiscoveryService` for locating and loading languages, whose files, such as the parse table used for syntax coloring, are stored in directories or compressed archives. Once loaded, a language is represented by one or more *language implementations*, essentially representing the different versions of a language, of which one is currently active. A language implementation aggregates the language's components, where each component contributes a number of *facets* to the language. Facets describe language services such as the language's parser and analyzer.

The existing implementation of Spoofax Core in Eclipse installs a *Spoofax* text editor component in the editor, which transparently handles the various Spoofax languages under the hood. This way, Eclipse can remain oblivious to the actual Spoofax languages and need not deal with their dynamic loading and unloading. The default behavior of Eclipse is overridden for Spoofax, such that whenever the IDE invokes an editor service on a Spoofax document, for example, applying syntax coloring to the code, Spoofax Core takes over. It then uses the document's filename and file extension to determine the actual Spoofax language used, and calls the corresponding language service.

Issues

IntelliJ, while also a Java-based editor like Eclipse, has a very different architecture. One aspect is that its editor services are tightly linked to the language of the source file, such that in some editor services, for example syntax coloring, IntelliJ does not provide the service with the filename of the source document, as it is assumed the service is language-specific and can work without knowing the filename. This precludes the use of a generic *Spoofax* language for IntelliJ, as used in Spoofax Core for Eclipse, where the editor services determine the actual Spoofax language from the filename.

A solution would be to load each Spoofax language as a separate IntelliJ language, but it is not easy to perform dynamic language loading while the IDE is running. IntelliJ plugins that contribute a language have to advertise this in their `plugin.xml` plugin configuration file, which is read by IntelliJ when the plugin is loaded. IntelliJ loads plugins only when the IDE starts, ensuring all contributed languages are loaded before the user starts editing.

Furthermore, IntelliJ expects every language to be represented by a class derived from the `Language` class, which IntelliJ uses to keep track of the languages and their attributes, such as the file types associated with it. IntelliJ distinguishes languages based on the *class* of this `Language` object, not just the object instance itself. Therefore the class representing a language must be different for every language. This is normally not an issue since every language plugin will have for each language a class derived from the `Language` class, but is an issue for Spoofax, since we do not know the languages to be loaded in advance, and therefore cannot define their classes in advance.

Approach

To ensure all IntelliJ editor services know the exact Spoofax language being used, we load a separate IntelliJ language for each loaded Spoofax language, and associate the language with its language's file extension. This way, we use IntelliJ's built-in mechanism that uses the filename to determine the language. However, this required us to solve two problems: IntelliJ loads languages as part of plugins at startup, while we want to load and unload languages

while the IDE is running; and IntelliJ distinguishes languages based on their `Language` *class*, while it is impossible to know in advance which languages will be loaded and therefore which classes need to be defined.

We found a solution to the first problem, loading and unloading a language while IntelliJ is running, by invoking the registration functions that IntelliJ itself invokes when loading a language at startup or unloading a language at shutdown. A language, its file type, and its editor services are normally declared in the plugin's `plugin.xml` file as *extensions*, which are read and registered by IntelliJ upon loading the plugin. For Spoofax Core, we manually call IntelliJ's registration functions, `ExtensionUtils.register` and `FileTypeUtils.register`, whenever we want to load a language. By calling these functions within a protected region, meaning that IntelliJ cannot perform any writing actions and has to wait, we were able to load and unload languages while the IDE is running.

The second problem, where each language must be represented by a dedicated class, is solved by using Javassist[1], which is a library that can create classes and compile code at runtime. Through its `ProxyFactory`, we build a class inheriting from `Language` to represent each Spoofax language.

Evaluation

We were able to load and unload languages dynamically by creating their classes at runtime and calling IntelliJ's extension registration functions. It works seamlessly, and IntelliJ automatically reloads any editors for which we reload the language.

However, we found that this implementation requires maintenance and breaks easily whenever a new version of IntelliJ is released. In once case a new IntelliJ release refused to load our Spoofax plugin, stating that our plugin made unexpected changes to the state of the editor. Apparently changes in the new release were reading and writing the editor state where it has not done so before, causing a conflict with our calls to the application programming interface (API). The solution was to add additional protected regions before calling the registration functions. Similarly, as the registration functions are part of the undocumented internal API of IntelliJ, future IntelliJ releases may change their semantics, or replace them entirely, breaking our plugin.

2.1.2 Syntax Coloring and Parsing

Syntax coloring is the editor service that colors the terms in source code according to their syntactic category, to improve the readability of the code. Most languages use a lexer-parser combination to interpret the source file. The lexer is used to split the source code into a stream of tokens, where each token describes a small part of the source code, such as an identifier or an operator symbol. The category of the token determines the color and styling, while the token stream is fed to the parser, which uses it to build an abstract syntax tree (AST).

Spoofax Core provides three services that together provide syntax coloring: the `SyntaxService` that returns the AST and tokens of a source file, given the file's language, filename, and content; the `CategorizerService` that associates each token with a syntactic category; and the `StylerService` that returns the color and font styling for each category.

Syntax coloring may take some time, during which the editor's user interface (UI) must not freeze. Therefore, in Eclipse syntax coloring is performed as an asynchronous job that is scheduled whenever the editor component's content needs to be recolored, such as when the user changes the text. When the job is run, it invokes the Spoofax Core services and uses the results to create an Eclipse `TextPresentation` object that describes the style of the source code, which is then applied to the current editor.

[1] `http://www.javassist.org/`

Issues

The IntelliJ architecture assumes that any language has an *incremental* lexer, meaning that when the source file has changed, the editor determines the region that needs to be updated and asks the lexer only for the tokens in that region. However, Spoofax Core does not have the distinction between a lexer and a parser: its `SyntaxService` uses the Spoofax SGLR scannerless parser, a lexer-parser combined into one, producing an AST directly from the source text. This allows the parser to deal with more complex grammars, such as grammars that contain ambiguities. We have to make the syntax service work with IntelliJ's lexer-parser paradigm.

Spoofax supports *concrete syntax extensions*, where a new language dialect is created that extends the host language's syntax, to allow code fragments to be written in the concrete syntax of an object language. The dialect to use for a particular file is specified in a separate file with the same name but with the `.meta` extension. Therefore, Spoofax needs the filename and path of the file being parsed to find the `.meta` file and be able to determine the language dialect to use. However, the standard API of IntelliJ assumes that a language's lexer and parser can parse a file's text without knowing the filename or file extension, something which is true for most languages but not for Spoofax languages.

Finally, IntelliJ uses data structures that make it easier to contribute editor services from scratch for a new programming language, by removing most of the boilerplate code, but this makes it harder to adapt Spoofax Core's existing services, as Spoofax Core does not use these IntelliJ-specific data structures. One such structure is the Program Structure Index (PSI) tree, an index IntelliJ maintains with the AST of all source files, and metadata about the project and its files. The ASTs returned by the `SyntaxService` will have to be adapted to work with the PSI tree of the opened project.

Approach

For IntelliJ we created a non-incremental lexer that always returns tokens for the whole file, by extracting the tokens from the AST returned by the Spoofax Core `SyntaxService`. To simulate an incremental lexer, we only return those tokens that intersect with the region IntelliJ asked for.

Each token returned by the lexer has an associated *token type*. The token type is an object that IntelliJ uses to look up the coloring of the token. This implies that if two tokens have a different style, they must have a different token type. The editor supports only up to 15.000 different token type objects in a single language, so while the naive implementation would create a new token type object for each token, we needed to cache our token type objects and only create a new token type when no existing cached token type covers the given style.

For the IntelliJ parser we again used the `SyntaxService`, and as it already returns an AST, we only had convert from this AST to the IntelliJ AST, where each node spans a range of tokens in the source file, and has an *element type*. Different element types have different features, such as an identifier type that can support reference resolution, which we determine through the Spoofax Core `CategorizerService`. IntelliJ automatically builds and adds a PSI representation of the AST.

By overriding IntelliJ's lower-level parsing API, we have more control over the parsing process and access to the filename and location of the file being parsed. This solves the issue of IntelliJ withholding the filename, which we need to determine the language dialect.

Evaluation

Through the above approach we were able to adapt the Spoofax Core syntax service to the IntelliJ lexer-parser model. Our implementation updates the coloring of a source file as expected, whenever its content changes or when the language of the file changes (when the language is dynamically reloaded).

Figure 2.1: Incomplete syntax coloring in Spoofax for IntelliJ. The left image shows the syntax coloring after typing the `/*` symbol that starts a comment, where only the next line is colored as a comment. The right image shows the expected coloring, where all following lines are colored as comments.

We found that changes influencing a larger part of the coloring of the code are sometimes not correctly shown. For example, starting a *block comment* near the top of the document should color the rest of the document as a comment. This is also what happens in Eclipse, but in our IntelliJ implementation sometimes only the coloring on the changed line is updated, as shown in Figure 2.1. It is not clear whether this is because IntelliJ expects the lexer to be incremental, which the adapted Spoofax Core syntax service is not, or because of some other reason.

Another consequence of the non-incrementality of the syntax service is the impact on performance. Even a small change will cause the syntax service to parse the entire document and produce an AST. When the document is large, this causes a noticeable delay in how fast the characters appear, but we have not measured this exactly. This delay is not present in Eclipse, suggesting that the synchronous implementation of syntax coloring in IntelliJ is causing the slow down.

Finally, most editors, including Eclipse and IntelliJ, support setting a dark theme for the editor UI. When a dark theme is used with a Spoofax language, the coloring of the source code does not change to match the dark theme, making the text hard to read. For example, black text that was perfectly readable on a light background becomes near invisible on a dark background, and this is an issue in both IntelliJ and Eclipse. The reason is that Spoofax does not support changing the token colorization based on the current editor theme. One possible solution would be to use categorize the tokens according to their syntactic role, and leave it to the editor to determine the colors to apply for tokens in each category.

2.1.3 Project Configuration

Spoofax language projects used to have their configuration stored in Maven project configuration files. Maven is a build system for Java programs that uses Extensible Markup Language (XML) files to configure the build. The configuration includes the settings for the parser and analyzer, and the languages on which the project depends.

Issues

When porting Spoofax Core to IntelliJ, we ran into the fact that IntelliJ's build system bundles its own specific version of the Maven dependencies, whereas Spoofax Core uses a different version of the Maven dependencies to read the configuration files, which caused dependency resolution errors in our plugin. A solution would either use a technique to hide the Spoofax Core Maven dependency from IntelliJ, or replace Maven with something else entirely.

Also influencing this choice was the desire to reduce the number of Spoofax Core dependencies, to improve maintainability and reduce the number of dependencies that can cause

breakage or security issues. As using Maven for configuration added a total of 28 (transitive) dependencies to Spoofax Core (Apache 2015), we chose to replace Maven and overhaul the Spoofax project configuration system.

Approach

Replacing Maven meant we had to move the Spoofax project configuration from Maven's XML build configuration files to a separate file, and we could pick a syntax to use for this new configuration file. As Spoofax language artifacts, the archive that bundles all files of a language together, already contained a descriptor file written in the YAML Ain't Markup Language (YAML) language, we chose to use YAML for the new project configuration as well.

YAML is a platform-agnostic language for describing data, usually configuration data, whose main objective is to be human readable. The language is similar to JavaScript Object Notation (JSON), so similar in fact that the latest version of YAML is a superset of the JSON language. (Ben-Kiki 2009)

We rewrote the Spoofax Core configuration services as part of the transition to YAML configuration files to be more flexible and extensible. Spoofax has configurations for projects, language components, and language specifications, each of which have a corresponding `ConfigService` that retrieves a configuration; a `ConfigBuilder` that builds a configuration; and a `ConfigWriter` that writes the configuration back to file. An example of a Spoofax project configuration written in YAML is shown in Figure 2.2.

```
---
id: org.zamenhof:Esperanto:1.0.0-SNAPSHOT
name: esperanto
dependencies:
  compile:
  - org.metaborg:org.metaborg.meta.lang.esv:${metaborgVersion}
  - org.metaborg:dynsem:${metaborgVersion}
  source:
  - org.metaborg:meta.lib.spoofax:${metaborgVersion}
  - org.metaborg:org.metaborg.meta.lib.analysis:${metaborgVersion}
language:
  sdf:
    pretty-print: esperanto
  stratego:
    format: ctree
    args:
    - -la
    - stratego-lib
    - strc
```

Figure 2.2: Example of a Spoofax project configuration written in YAML.

Evaluation

Through the transition of Spoofax project configurations from Maven's XML files to YAML files, we reduced the number of configuration-related dependencies from 28 to only eight. As Maven is no longer a dependency of Spoofax, the version conflict with the Maven version used in IntelliJ's build system is resolved.

2.1.4 Reference Resolution

Spoofax Core supports reference resolution, resolving a reference in the source code to its definition, through the `ResolverService`. Reference resolution is supported in IntelliJ, next to symbol navigation services such as *Go to Symbol* and *Find Symbol*, but these are not supported by Spoofax Core.

IntelliJ maintains metadata about the project and ASTs of the source files in its PSI index, and uses this index to provide reference resolution capabilities. Reference resolution is implemented in our plugin by augmenting the IntelliJ PSI elements with a method that returns these references. Since the Spoofax Core reference resolver service can arbitrarily decide that a term is a reference, *all* PSI elements created by the parser pretend that they could have a list of references, and they all ask the service to resolve references. Only when the resulting list of references is non-empty, will the IntelliJ editor resolve a reference by jumping to its definition.

2.1.5 Transformations

A Spoofax language can contribute *builders*, which are transformations that are applied to a single source file. For example, a builder commonly found in Spoofax languages is automatic formatting, which changes the source code to have a consistent formatting of whitespace and newlines.

In Eclipse, the builders that are contributed by the Spoofax language of the active source document are shown in a menu. As different languages contribute different builders, the menu is updated whenever the active document changes.

In IntelliJ we recreate this builder menu. It would be very expensive to re-instantiate the menu and menu items every time the user selects a different file to edit. Instead, we cache the menu and menu items, selectively make the items visible, and modify them to show the builders for the current language.

2.1.6 New Project Wizard

The Spoofax plugin in IntelliJ supports two kinds of projects: Spoofax language definitions, and Spoofax projects. A language is defined in a Spoofax language definition, and used in a Spoofax project. Spoofax projects are generally simpler, because they do not need the infrastructure for a full fledged language definition.

In IntelliJ a project is called a *module*, and each project has an associated *module type* that determines the kind of project, such as a whether it is a Java or Scala project. For Spoofax language definitions we created our own *Spoofax* module type, which allowed us to hook into the project-specific configuration and build process.

We wrote a *wizard* to enable users to create a new Spoofax language definition project, shown in Figure 2.3. The wizard is the standard IntelliJ wizard for new projects, with some added configuration fields, where the user is asked to specify the name, file extension, group and artifact identifiers, and artifact version of the language. The default values for most of these fields are derived from the name specified by the user. The wizard also allows the user to set the domain-specific languages (DSLs) to be used defining the syntax and name resolution.

Almost all languages have some boilerplate files and folders they need for a project to function. At minimum, this includes configuration files and default folders for the source code. These files are automatically generated by the wizard when the user finishes it.

When an existing Spoofax project is imported into IntelliJ, it needs to be recognized as such to enable the Spoofax-specific functionality. We implemented this in a *project detector service*, which determines whether a project is a Spoofax language specification project by looking for the `metaborg.yaml` project configuration YAML file in the project's root folder.

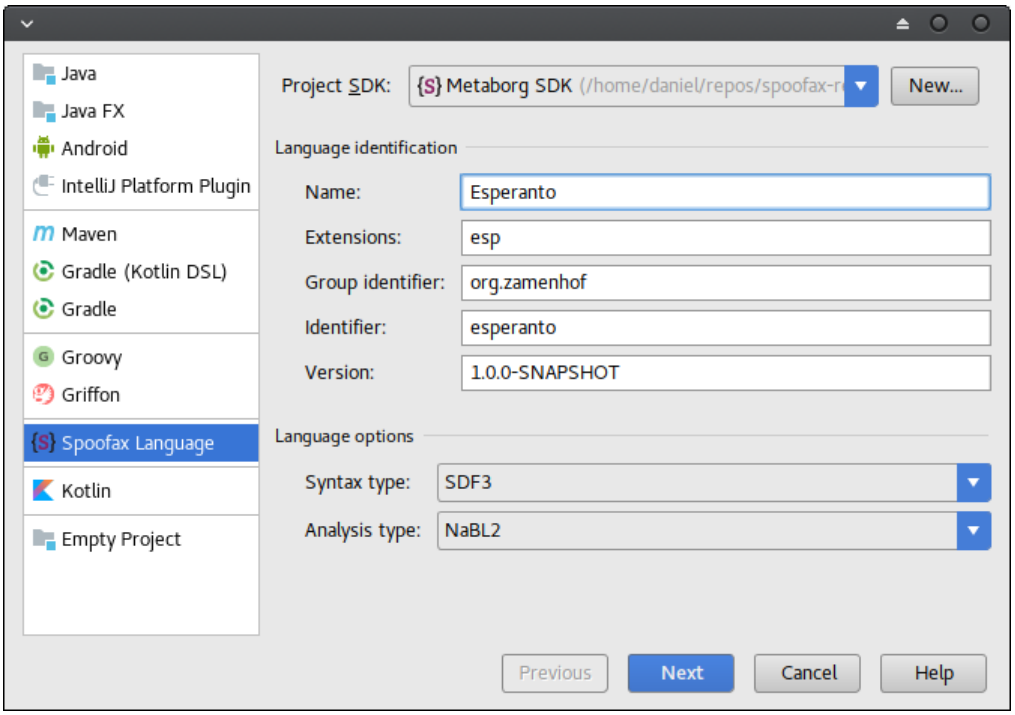


Figure 2.3: Wizard for creating a new Spoofax project in IntelliJ, asking the user for a language name, file extension, and other identifiers. The user can pick which DSLs to use for defining the syntax and name resolution.

2.1.7 Build System

IntelliJ builds projects using a separate build system called JetBrains Project System (JPS), developed to be independent of the IDE. JPS builds its own model of the project, where the configuration of the application, project, and its individual modules are passed from IntelliJ to JPS through a serializer. For Spoofax Core, this configuration is a list of the specific metalanguages that should be loaded for the source code to be compiled.

JPS represents the build process as *build targets*, which are actions that need to be performed on the project files. Build targets can depend on other build targets, forming an acyclic directional graph. JPS determines the order in which build targets are executed, and independent build targets can be executed in parallel.

The build process of a Spoofax language definition is represented by three build targets: the Spoofax pre-builder, the Java builder, and the Spoofax post-builder. The first target is the Spoofax pre-builder, which initializes the Spoofax Core compilation process, generates any source files that need to be generated, analyzes and transforms the files from the project, and performs some final tasks to setup the Java files. The second target is the built-in Java builder, which compiles all Java files into class files and is part of JPS by default. The third and final target is the Spoofax post-builder, which gathers the class files and related resources and packs them into an artifact, ready to be used as a language.

The JPS build system has dependencies of its own, whose versions sometimes conflict with the same dependencies used by Spoofax Core, such as is the case with the Maven dependency and a dependency on Apache Commons IO. We removed the Maven dependency from Spoofax Core, and found two ways around this for Commons IO: either implement a custom Java class loader; or alter the package that contains the conflicting dependencies.

Java class loaders are responsible for finding the correct Java class given a class name, and loading it into the Java virtual machine. We tried to make a class loader that can distinguish between the same dependency for Spoofax Core and JPS, and load the correct version of the

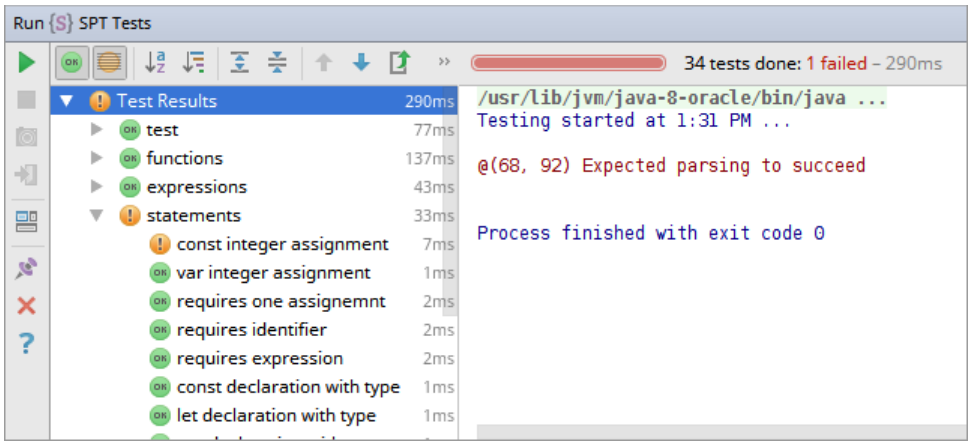


Figure 2.4: The Spoofax test runner in IntelliJ shows the tests and their current state.

dependency for either of them. However, after much experimentation this proved to be near impossible, as our class loader conflicted with the custom class loader used by JPS.

The second option was to pack the conflicting dependencies into a fat JAR, an archive which contains the compiled Java files. We could then rewrite the package names of these dependencies to not clash with those used by JPS. This is the actual solution we used.

2.1.8 Diagnostic Messages

Diagnostic messages, the error and warning messages that are shown as a result of issues with the code, are provided by an *annotator* in IntelliJ. The annotator analyzes the source code and produces warning and error messages, such as when a reference could not be resolved, or when there is a duplicate definition. The Spoofax `AnalysisService` can take arbitrarily long to analyze the source code, so we implemented our annotator as an *external annotator*, which IntelliJ runs in a separate thread to avoid blocking the main user interface.

2.1.9 Unit Test Runner

Language developers can write declarative tests for their Spoofax languages through the Spoofax Testing language (SPT). Such tests can verify whether services such as the parser, analyzer, and code generator produce the expected results, such as a concrete code fragment, AST, reference resolution, or error message.

IntelliJ's integrated test interface can show the available tests and the results for each of them. Its default implementation listens to messages from JetBrains' TeamCity continuous integration service. We integrated the SPT test runner with IntelliJ by adapting the test runner to produce these TeamCity status messages on the standard output. These messages indicate when a test suites is entered or exited, when a test starts running, and what the result of running the test is. IntelliJ interprets the messages and displays the tests and their state to the user, as shown in Figure 2.4.

2.2 Conclusion

Implementing Spoofax for IntelliJ revealed some bugs and required some changes to Spoofax Core. The biggest impact on Spoofax Core was replacing the Maven-based configuration with our extensible JSON project configuration format.

Spoofax Core is platform-agnostic, as we have shown by porting Spoofax Core to IntelliJ. However, Spoofax Core continues to change and evolve. Bugs get fixed, support for new

meta-languages such as the FlowSpec data-flow analysis language or new tools such as the JSGLR2 parser are added, and research projects such as Spoofax PIE pipelines will overhaul the existing Spoofax Core architecture to support incremental editor services and build services. Even if no new features are added to the various editor implementations of Spoofax Core, they will have to be changed to accommodate the changes to Spoofax Core itself. This is an additional maintenance burden.

With the development effort concentrated on the Eclipse implementation, and it being the more mature implementation of the two, the IntelliJ workbench remains relatively unused. This in turn causes many to prefer the Eclipse workbench over the IntelliJ workbench, especially when they encounter a bug and few are versed enough in the IntelliJ implementation to be able to fix it.

Maintaining more implementations is an extra effort. When there is a more stable alternative (Eclipse), the other implementations are easy to neglect. Spoofax Core continues to change, but the IntelliJ implementation is not always correctly updated.

If we could abstract the complexities of and changes to the Spoofax Core model from the adapters to the various editors, then changes to Spoofax Core need not impact the adapters to each editor as much, if at all. In Chapter 3 we explore how a simple language and associated editor services could be implemented in IntelliJ and Xtext. In Chapter 4 we look at how editors expect editor services to be implemented. Then in Chapter 5 we design a generic model for editor services based on what we learned, providing a solution to this maintenance and portability problem.

Chapter 3

Editor Plugins

Most editors and integrated development environments (IDEs) can be extended to support additional programming languages through an editor plugin, which provides an implementation for the language and its editor services. To develop portable editor services, we need to explore how editor services work in other editors, and the support editors and language workbenches provide to aid their implementation.

We explore editor services by writing an editor plugin for the IntelliJ IDE, and an plugin using the Xtext language workbench. Both assist in the creation of editor services for a new language, albeit in different ways.

The IntelliJ IDE, unlike other editors, has data structures and boilerplate services on top of which a language's editor services can be built. We have not explored these in our implementation of Spoofax Core in IntelliJ (Chapter 2), as Spoofax Core already provided these editor services itself.

Xtext (Efftinge and Völter 2006) is a language workbench built around the Xtext declarative language specification and the Xtend Java dialect. It generates most editor services from a declarative specification of the language, and this allows the workbench to support multiple editors, including the Eclipse and IntelliJ.

We implement parts the Programming and Programming Languages in Java (PAPLJ) language, which is a small Java dialect used as an exercise in the online book Declare Your Language (Visser 2015)[1], and loosely based on the language used in the online book Programming And Programming Languages (Krishnamurthi, Lerner, and Politz 2017)[2]. A program in PAPLJ is represented by a single source file that defines classes with fields and methods, and a *run* expression to be evaluated. PAPLJ has expressions, but it has no concept of statements. For example, while the conditional `if` is a statement in normal Java, it is an expression in PAPLJ. The grammars in Appendix C specify the syntax of the language.

This chapter continues as follows: in Section 3.1 we explain how a language plugin is generally implemented in IntelliJ, and what functionality IntelliJ provides to support this. In Section 3.2 we do the same for the Xtext workbench, and use that to produce a plugin for Eclipse.

3.1 IntelliJ Language Plugin

We found that there are two ways to set up a plugin project for IntelliJ: as a special IntelliJ plugin project in the editor, or as a Gradle project that uses the official `gradle-intellij-plugin`[3]. IntelliJ supports either equally well, but we chose the solution using the Gradle build system

[1] `https://github.com/MetaBorgCube/declare-your-language`

[2] `http://papl.cs.brown.edu/2017/`

[3] `https://github.com/JetBrains/gradle-intellij-plugin`

as it additionally allows the plugin project to be compiled and run from the command-line or any editor that supports Gradle projects.

Setup After setting up the project, the very first thing is to define our own language and corresponding file type. A language in IntelliJ is a class extending the `Language` class, that defines a unique identifier for the language. The identifier is used to refer to the language in the plugin's `plugin.xml` file when registering extensions and services for the language.

The first extension we had to register is an extension of the `FileTypeFactory`, which tells the editor to associate our custom file type class with PAPLJ source files, for which we chose the `.pj` extension.

Program Structure Index IntelliJ internally represents a project as a big structure called the Program Structure Index (PSI). The PSI stores data about the modules in the project, the content roots in each module, and the files and folders in each content root. For each file, the PSI includes the abstract syntax tree (AST) of the file. IntelliJ uses the PSI structure to provide quick responses and a nicer user experience. For example, when the user places the caret on an identifier in the source code, IntelliJ can highlight its corresponding PSI element immediately, without having to ask the plugin whether the text under the caret is an identifier and if so, where it starts and ends. Plugins can also take advantage of the PSI representation, for example to walk up the tree to resolve a reference to its definition. (JetBrains 2018h)

Syntax Coloring To enable syntax coloring, the language implementation has to build the PSI structure for the source file. IntelliJ expects a language plugin to provide a lexer and parser to build the PSI structure. A lexer reads the source code input and produces a stream of tokens, such as keywords, comments, operators, and identifiers, and feeds them to the parser. The parser then builds an AST with the tokens as the leaves.

IntelliJ requires the lexer to produce a stream of PSI element types, each describing one token in the source code input. Tokens of the same syntactic meaning must be represented by the same PSI element type, as IntelliJ uses these to determine the coloring of each part of the source. IntelliJ then uses the element type to create a PSI leaf element to represent each token, and feeds these to the language's parser. The parser then has to build a tree of PSI elements.

Writing all the classes for the PSI elements, element types, and the IntelliJ-specific parser and lexer by hand is a large effort. Fortunately, IntelliJ includes the *Grammar-Kit* tool which can generate these classes for us, when given a lexer and parser grammar definition.

The lexer and parser grammars must be written using the JFlex and Backus–Naur form (BNF) syntaxes respectively. JFlex is a popular lexer generator for Java, and IntelliJ uses it to generate a lexer that produces PSI element types instead. The BNF format is specific to Grammar-Kit and uses a Backus–Naur form-like notation to describe a parsing expression grammar (PEG) (which is similar to context-free grammars, but no ambiguities can occur), from which an IntelliJ-specific recursive-descent parser is generated.

The BNF format forced us to rewrite the PAPLJ grammar to have separate productions to explicitly control operator precedence and associativity. Another issue was PAPLJ's Java-like cast syntax `(Foo)x`. The parser could not distinguish between the cast and an identifier in parenthesis. Therefore we changed the syntax to `x as Foo`. Our lexer and parser grammars for the PAPLJ language can be found in Appendix C.3.

Editor Services We have not implemented other PAPLJ editor services in IntelliJ. Editor services such as reference resolution and code completion require implementing name and type analysis for the language, something which is not easy to do and which is not specific to IntelliJ. Also, we would not learn much from binding this analysis to our IntelliJ plugin,

as their implementation requires very little code due to everything being defined in terms of the PSI structure we already built.

Conclusion The PSI structure, with the ASTs of the source files, permeates every editor service in IntelliJ. With the editor maintaining the in-memory representation of the project and its files, IntelliJ can take control of optimizations such as whether editor service results are cached, which makes for the highly responsive editor IntelliJ is known for. This is great for language plugins that are written from the ground up, as they can fully utilize the tools IntelliJ has developed for working with their internal representations, such as the PSI structure. However, as we saw previously in Chapter 2, for an existing language it puts the burden on the language developer to adapt the language's structures to those of IntelliJ.

The repository for PAPLJ in IntelliJ can be found online at `https://github.com/Virtlink/paplj-intellij`.

3.2 Xtext Language Plugin

Like Spoofax, Xtext is a textual language workbench (Efftinge and Völter 2006). It is part of the Eclipse Modeling Framework Project, and as such supported by Eclipse. The workbench relies on the ANTLR LL(*) parser generator (Parr and Quong 1995), and supports syntax coloring, diagnostic messages, code-completion, and automatic formatting in Eclipse, IntelliJ, browser-based editors, and Language Server Protocol (LSP) (Microsoft 2018a).

Project Through the Xtext project generator, we generated a bare project. The generator asks for the language name and file extension, and which facets you want to add. We added the Eclipse, IntelliJ, LSP, and *Web Integration* facets, which allows us to run our language in those editors with minimal effort. The generator produces a number of mostly empty projects: a language definition project, an editor services project, and a project for each of the editors. We only worked with the language definition project.

Syntax The primary file in any Xtext language definition is the language's `.xtext` file. This file is written in the Xtext declarative language, which as a syntax similar to that of Another Tool for Language Recognition (ANTLR), and is where the syntax of the language is defined. We readily adapted the grammar from IntelliJ to Xtext, as Xtext also requires explicit precedence and associativity. In an Xtext grammar, identifiers that are used as references can specify their target. For example, the type of a field in PAPLJ is specified as a `QualifiedName` that refers to a `Type`. This information is used for reference resolution, code completion, and the structure outline.

Scopes The next step was to define the scoping rules, in which a scope determines where a declared name is visible. For example, a method scopes the variables defined within it, as they are only visible to other code within the same method. In Xtext the scoping rules are defined by an implementation of the `IScopeProvider`, which when given the referencing element and its context element, returns a scope object which knows all the names of all the objects that the reference can refer to. We implemented the scope provider such that member and variable references are scoped to their respective class and method.

Types and Type Resolution With the scoping rules in place, we needed some core types for our language. We chose to support only boolean and numeric value types, represented by `Bool` and `Num` respectively. Furthermore, we needed a type to represent the object at the top of the hierarchy, and call it `Any`. The easiest way to let Xtext know about these pre-defined

types is by defining them in a separate PAPLJ source file that is automatically imported with any other PAPLJ files.

With predefined types and proper scoping, we could tackle the type resolution. Xtext made this easier, automatically resolving identifiers in the source code to the elements being referenced. We defined our own function `typeOf` that returns the types of the various expressions in PAPLJ. Since we only have one kind of number, determining the type of the operators was simple, as they all either always return a boolean value or a number value. The *new* and *cast* expressions were also trivial, as their type is explicitly encoded in the source code. Only the variable and member references were more complex, as we have to look up the declaration being referenced, for which Xtext does most of the heavy lifting, and determine the declaration's type. Finally, we provided some functions to determine whether the actual type of an expression conforms to the expected type, taking the class hierarchy into account.

Validation Using the type information, we were able to write some validation rules. Our validation rules check for duplicate names, incompatible types, cycles in the class hierarchy, and field that are used as methods and vice versa.

Generator Finally, we wrote a Java code generator for PAPLJ in Xtext, where the generator walks through the AST and outputs the corresponding Java code. However, the generated code is not entirely correct. For example, the *let* expression is compiled as a Java statement, which is wrong as this precludes the use of the *let* in any expression context. Also, for the *run* expression a `main` method should be generated.

Conclusion Xtext primarily supports Eclipse, where our editor services worked out of the box. Xtext loads our language in a separate Eclipse instance, which we used continuously to debug our definition.

We also launched the generated PAPLJ language server. For this we needed an extension for VS Code to act as the language client. This extension is not generated by Xtext, so we had to write it ourselves. The language server has no syntax coloring support, but reference resolution and diagnostic messages worked as expected.

For the web-based editor for our language, a server process is launched to act as the local server. The editor is based on the Ace web editor[4] and quite minimal: it only supports syntax coloring, diagnostic messages, and code completion. Any other support, such as saving changes, has to be implemented manually.

Xtext boasts in its documentation support for IntelliJ, but when we tried it we got multiple errors. Further investigation led to a statement by the developers that they no longer support IntelliJ due to a lack of resources, and hope for IntelliJ to support their language server instead. (Ogarkov 2017)

Xtext is great when one can declare the language from scratch, utilizing the tools and domain-specific languages of Xtext to their fullest. Xtext takes care of generating the code for the various platforms it supports. However, existing language implementation can not be adapted to Xtext to gain support for the platforms Xtext supports, as the application programming interface (API) is internal to Xtext.

The repository for PAPLJ in Xtext can be found online at `https://github.com/Virtlink/paplj-xtext`.

[4] `https://ace.c9.io/`

Chapter 4

Comparing Editor Services

To design a generic editor services interface, we first needed to examine how a language can implement editor services in various editors. Ideally, one can use a single programming language in many different code editors. By catering to the different editor needs and preferences of developers, the barriers to entry of using the language are lowered and this will likely have a positive effect on the overall adoption of the language. To achieve this, the language will need to adapt its implementation to the editor services of the various editors it wants to support.

Editors that allow their editor services to be programmatically implemented, as opposed to statically declared ahead of time, often expose an application programming interface (API). This is a specification of the function calls, data structures, and object classes that the application can accept and return. As long as an implementation adheres to the API specification, it will work with the application. Code editors tend to have similar editor service APIs, as they present their services in similar ways to support various languages.

Our contribution in this chapter is an exploration of the editor services of the most popular code editors, where we look at the features they have and the requirements they impose on language implementations. We do this by examining the ways in which editors allow their editor services to be extended, either through an API or a declarative specification. While details such as the implementation language or interoperability protocol may vary greatly between code editors, having a generic idea of the requirements of editor services in the various editors should reduce the effort required to port an editor service implementation. We will use what we learned to drive the design of our editor services interface in Chapter 5.

We used the Stack Overflow Developer Survey of 2017 (Stack Overflow 2017) to determine which editors to compare, and picked the top eight most popular developer environments. These are, in order of popularity in the survey: Visual Studio, Notepad++, Sublime, Vim, Eclipse, IntelliJ, VS Code, and Atom. This is a healthy mix of full-featured integrated development environments (IDEs), those that at least include build automation and debugger support, and simple source code editors, some of which can be augmented with IDE-like features through plugins. Out of all respondents to the Stack Overflow survey that answered this multiple-choice question, more than 94% picked at least one of these editors.

While browser-based cloud applications steadily grow in popularity, browser-based code editors were not part of the Stack Overflow survey. Their advantages, such as low platform requirements and immediate updates to the latest version of the editor, make these editors good candidates for developers with low-power tablets and ultrabooks. Therefore we also include two cloud-based IDEs: Cloud9 and Eclipse Che (Slant 2018).

We focus on those editor services that are most relevant to language users yet implementable through language definitions. Erdweg et al. (2013) identified a feature model for language workbenches; the most important features for language workbenches to have. From this model we derived thirteen editor services we can expect to find in the code editors we

investigate, namely: syntax coloring, code folding, code completion, structure outline, reference resolution, hover documentation, signature help, automatic formatting, rename refactoring, code actions, diagnostic messages, debugging, and testing.

Table 4.1 provides an overview of the support of the selected editor services in the code editors we investigated. Six out of ten code editors have an API for most editor services, and those are also the editors that can be considered IDEs or IDE-like. The remaining four editors are simpler source code editors and lack many of the editor services. For the services that they do provide, many have only declarative support or the editor service can only be implemented through plugins.

	VS	NP++	Sublime	Vim	Eclipse	IntelliJ	VS Code	Atom	Cloud9	Che
Editor										
Syntax coloring	●	●	◒	◒	●	●	◒	◒	◒	◒
Code folding	●	●	○	◒	●	●	○	○	◒	●
Code completion	●	○	●	●	●	●	●	○	●	●
Navigation and structure										
Structure outline	○	○	○	○	●	●	○	○	●	○
Reference resolution	●	○	◒	◒	◒	●	●	○	●	●
Help and documentation										
Hover Documentation	●	○	○	○	●	●	●	○	●	●
Signature help	●	○	○	○	○	●	●	○	●	●
Actions and transformations										
Automatic Formatting	●	○	○	○	●	◒	●	○	●	●
Rename refactoring	●	○	○	○	●	●	●	○	●	●
Code actions	●	○	○	○	●	●	●	○	○	●
Life cycle										
Diagnostic messages	●	○	◒	◒	●	●	◒	○	●	●
Debugging	●	○	○	○	●	●	●	○	●	○
Testing	●	○	○	◒	●	●	◒	○	◒	◒

Table 4.1: Editor service support and programmability in popular code editors.
(●, programmable; ◒, supported but not programmable; ○, not natively supported)

This chapter continues as follows: Sections 4.1 to 4.6 compare the syntax coloring, code completion, structure outline, reference resolution, code actions, and debugging editor services across editors; Appendix A covers the remaining seven editor services. Section 4.7 takes a quick look at some editor services that we have not extensively covered.

4.1 Syntax Coloring

Arguably the most important part of any code editor is the editing area itself. All editors we looked at support text-based code editing with syntax coloring. Syntax coloring, also less accurately known as *syntax highlighting*, is the process of coloring the various terms in the source code according to their syntactic meaning. Usually, at least keywords, identifiers, and literals get different colors to make them stand out against one another. The syntactic categorization of the various terms in a source file differs from language to language, and the actual color assigned to a category of terms differs from editor to editor. Figure 4.2 shows two possible syntax colorings of a small snippet of Programming and Programming Languages in Java (PAPLJ) code.

Editors take several approaches to syntax coloring. Relatively few editors expose an API for syntax coloring, most support only a declarative specification of their syntax coloring, such as through a TextMate grammar or editor-specific configuration file.

Figure 4.2: The same code snippet in VS Code styled by the pre-installed Quiet Light (on the left) and Dark+ syntax themes.

4.1.1 TextMate Grammars

TextMate is a well established code editor for the MacOS operating system that introduced the concept of *bundles*: a collection of code snippets, commands, macros, templates, themes, and language grammars that allows users to extend the editor with more features or new languages. Nowadays the TextMate repository[1] alone hosts over 200 languages bundles.

TextMate grammars are widely supported: of the editors we looked at only Notepad++ and Vim have no support for TextMate grammars. Eclipse and IntelliJ have third-party plugins that add support for TextMate grammars, and the other editors can either import TextMate grammars into their own formats, or read TextMate grammars directly.

A TextMate language grammar is a declarative specification of a language's keywords and constructs. Originally written in Apple's Property List (plist) format, there now exists a JavaScript Object Notation (JSON) syntax, an ASCII syntax (Apple 2010a) very similar to JSON, and an XML syntax (Apple 2010b).

Figure 4.3 is an excerpt of a TextMate grammar for the PAPLJ language. The grammar starts with defining the language name and file extensions, followed by a description of the syntactic elements of the language, which are matched using regular expressions, and assigns *scope names* to them.

A scope name is an identifier for a syntactic element, which is then styled by a syntax theme. It consists of one or more identifiers each separated by dots, where each additional identifier specializes the scope. For example, the scope name of an arithmetic operator in the PAPLJ language might be `keyword.operator.arithmetic.paplj`. A theme specialized for the PAPLJ language would assign a specific style to elements with the `keyword.operator.arithmetic.paplj` scope, whereas a more general theme would only assign a style to the more general `keyword.operator.arithmetic` scope of arithmetic operators. A very simple theme might only have a style for the `keyword` scope, styling all keywords and operators in all languages exactly the same. As shown in Figure 4.2, a light syntax theme might render most text as dark gray on white, but render keywords in blue, whereas a dark syntax theme could render the text as white on black with purple keywords instead.

TextMate grammars apply regular expression patterns to each line to find the scope names of the elements. Most patterns, such as the keyword patterns in Figure 4.3, are quite simple and just assign a scope name to the result of the pattern match. However, it is also possible to assign different scope names to different parts of a regular expression. The most complex patterns match a start and end element that may be several lines apart, and apply other patterns to the text in between. The multi-line comment pattern is an example of such a pattern.

[1] `https://github.com/textmate/`

These patterns can even be recursive, such as a pattern that matches a (nested) class in Java, and this allows a grammar to be more accurate at the expense of simplicity.

4.1.2 Notepad++ and Vim

The only way by which Vim allows new language syntax colorings to be defined is statically through their own domain-specific language (DSL) *Vimscript*. Somewhat similar to TextMate grammars, one can define regular expressions, keywords, and regions for Vim to recognize, and specify their combinations. For example, a certain keyword may be followed by a number, whereas a similar string of digits in another context is treated as an identifier. Regions can also specify whether they are foldable. (Moolenaar 2011; Vim Tips Wiki 2017)

Notepad++, like Vim, supports defining syntax coloring for a new language through their own limited DSL, although it is less flexible. A Notepad++ user-defined language style is ultimately an XML file which specifies the language's keywords, identifiers, and operators, and delimiters for comments, strings and single characters. While this allows for ad-hoc coloring of many languages, the coloring is not flexible and can therefore easily go wrong in more complex situations, such as when a word is treated as a keyword in one context and as an identifier in another.

4.1.3 API

The only editors that provide a dedicated API for syntax coloring are Visual Studio, Eclipse, IntelliJ, and Notepad++.

Visual Studio

A syntax colorer for Visual Studio must implement a classifier that *tags* the terms of the source code with their classification type. Based on the classification type, the editor finds the corresponding classification format definition, which specifies the color and style of the terms. The relevant API is shown in Figure 4.4.

The source code is represented as a `ITextBuffer` object, which has methods to insert, delete, and replace text. The latest version of the text is accessible as a `ITextSnapshot` immutable snapshot with an associated version number. The associated `ITextDocument` text document object, if any, provides the filename of the source file and methods to save it.

The classifier, implementing the `IClassifier` interface, has a factory class `IClassifierProvider` which is registered with the editor. The classifier gets a region of source code, denoted by a `SnapshotSpan`, and must return a collection of `ClassificationSpan` objects. A classification span associates a `IClassificationType` with a small region of the source code, usually a single term. The name of the classification type is the unique identifier used by the editor to find a corresponding `ClassificationFormatDefinition` object, which provides the color and style. (Microsoft 2016i)

While language plugins can, and often do, provide their own classification formats, fixing the colors of the terms may not work well when the user changes the source code color scheme. Therefore Visual Studio provides a list of predefined classification type names in the `PredefinedClassificationTypeNames` class that, when used to classify terms, lets the language use the colors and styles specified in the current color scheme.

Eclipse

Eclipse records the styles of the source code terms in a presentation object. The usual way to create such a text presentation is through a system of presentation damagers and repairers, where upon a textual change the damager identifies the parts of the text that have to be recolored and the repairer produces the new coloring of this damaged part. Alternatively,

```
{
  "name": "PAPLJ",
  "scopeName": "source.paplj",
  "fileTypes": [ "pj" ],

  "patterns": [
    { "include": "#keywords" },
    { "include": "#numeric" },
    { "include": "#operators" },
    { "include": "#comments" }
  ],
  "repository": {
    "keywords": {
      "patterns": [ {
          "name": "keyword.other.paplj",
          "match": "\\b(program|run|class|extends|as|public|import)\\b"
      } ]
    },
    "numeric": {
      "patterns": [ {
          "name": "constant.numeric.integer.paplj",
          "match": "\\b\\d+\\b"
      } ]
    },
    "operators": {
      "patterns": [ {
          "name": "keyword.operator.comparison.paplj",
          "match": "(==|!=|<)"
      }, {
          "name": "keyword.operator.logical.paplj",
          "match": "(&&|\\|\\||!)"
      }, {
          "name": "keyword.operator.arithmetic.paplj",
          "match": "(\\+|\\-|\\*|/)"
      } ]
    },
    "comments": {
      "patterns": [ {
        "name": "comment.block.paplj",
        "begin": "/\\*",
        "end": "\\*/",
        "captures": {
          "0": { "name": "punctuation.definition.comment.paplj" }
        }
      } ]
    }
  }
}
```

Figure 4.3: Excerpt of the PAPLJ TextMate grammar. The full grammar is in Appendix C.2.

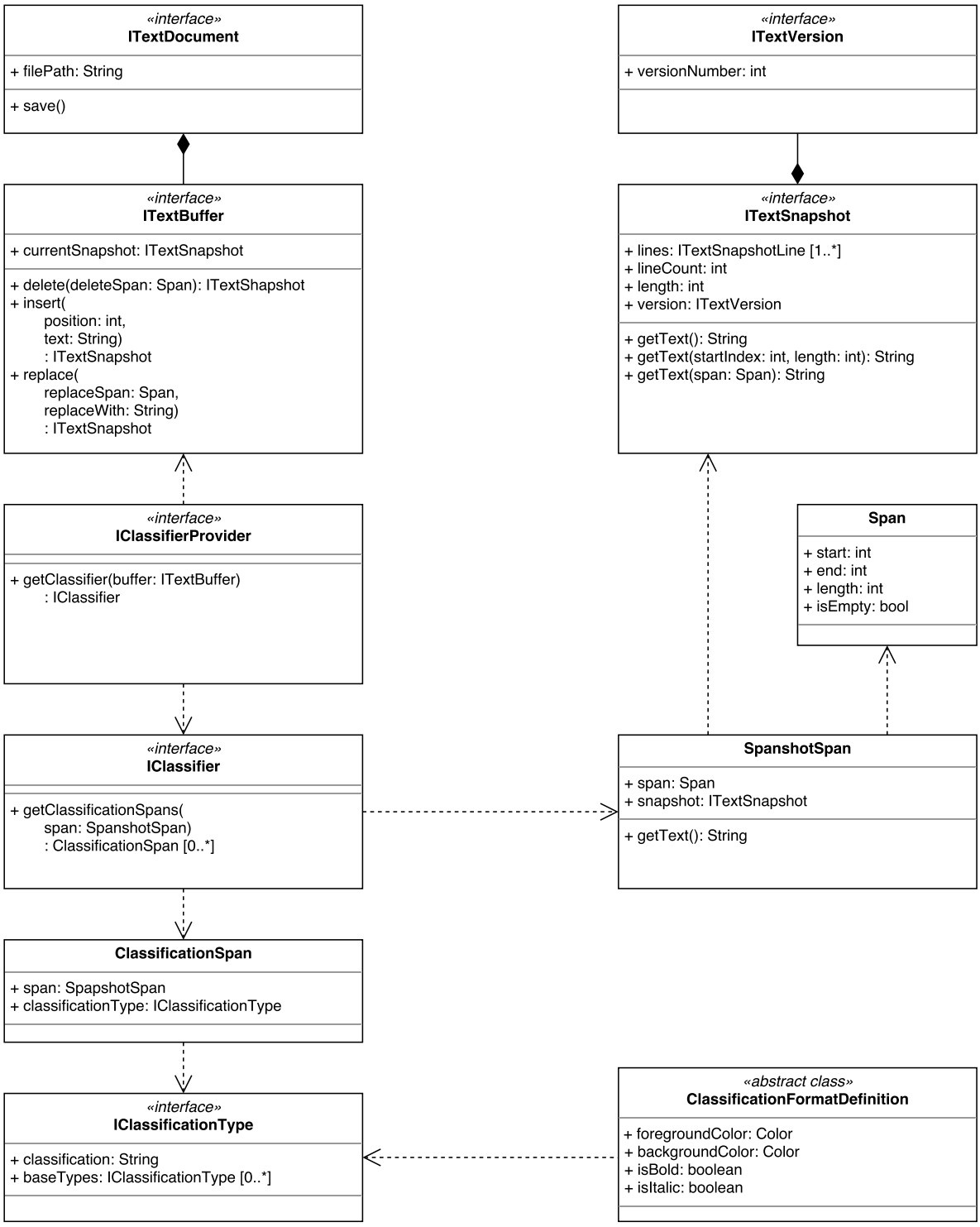


Figure 4.4: Overview of the API for the syntax coloring in Visual Studio.

the damager and repairer can be bypassed by applying the `TextPresentation` object directly to the editor component. The damager-repairer API is shown in Figure 4.5.

The `SourceViewerConfiguration` associated with the editor component specifies, among other things, the `IPresentationReconciler` object that is responsible for providing the presentation damager and repairer objects. The damager, an implementation of the `IPresentationDamager` interface, gets the region where the damage must be determine and the event that describes the change to the text, and must return the region for which the presentation has to be recomputed.

The `IPresentationRepairer` has to repair the text presentation, where the implementation adds style ranges to the `TextPresentation` object for the terms in the damaged region. A style range, represented by the `StyleRange` class, specifies the color, font, and style of a range of source code, usually a term. (Eclipse 2017a)

Eclipse does not have a built-in system for managing color schemes, but some plugins such as Eclipse Color Themes[2] can do this by changing the colors in the editor's preferences to match a given theme.

IntelliJ

IntelliJ represents each source files as its abstract syntax tree (AST), known as a Program Structure Index (PSI) tree in IntelliJ. The editor expects a plugin to build this AST from the source file, where the *element type* of the elements in the tree determines their coloring. The relevant API is shown in Figure 4.6.

A source file is represented as a tree of `PsiElement` objects, and each has an associated `IElementType` which determines the type of element and its coloring. The language's `ParserDefinition` must provide a lexical analyzer (lexer) and syntactic analyzer (parser), which IntelliJ uses to build the file's AST.

The lexer, an implementation of the `Lexer` class, is given a range of text and must produce a stream of tokens, where for each token the text, element type, and its range in the full text is specified. The editor calls the lexer's `advance` method to get the attributes of subsequent tokens.

IntelliJ expects the lexer implementation to be incremental: the lexer need only to reanalyze a smaller range of source code, and the editor will reconcile the resulting tokens with the full AST representation. To enable this, with each token the lexer returns its current state as an integer value. Whenever the editor calls the lexer again for a sub range, it passes the state the lexer had at the start of the range back to the lexer, so the lexer can continue from there.

It is the parser's job to create an AST from the tokens returned by the lexer. The parser, an implementation of the `PsiParser` interface, gets a `PsiBuilder` instance that interacts with the lexer. The parser calls the builder's `mark` method to mark the start of a subtree, which returns a `Marker` object on which the parser can call the `done` method to end the subtree. Calls to the builder's `advanceLexer` method in between calls to `mark` and `done` cause the lexer's tokens to be added to the subtree being built. The `done` method accepts an element type, which is stored in `ASTNode` objects from which IntelliJ builds the `PsiElement` AST.

Alternatively, the lexer-parser infrastructure can be bypassed by overriding the `doParseContents` method of the element type used to represent the language's files. This provides access to the `PsiElement` that represents the file being parsed, from which the filename and path can be determined. This is not otherwise possible using the built-in lexer-parser infrastructure.

With the AST built, IntelliJ calls an implementation of the `SyntaxHighlighter` interface, which provides the `TextAttributesKey` associated with a particular element type. The `TextAttributesKey` is used to look up the style information in a registry internal to the editor.

[2] `http://www.eclipsecolorthemes.org/`

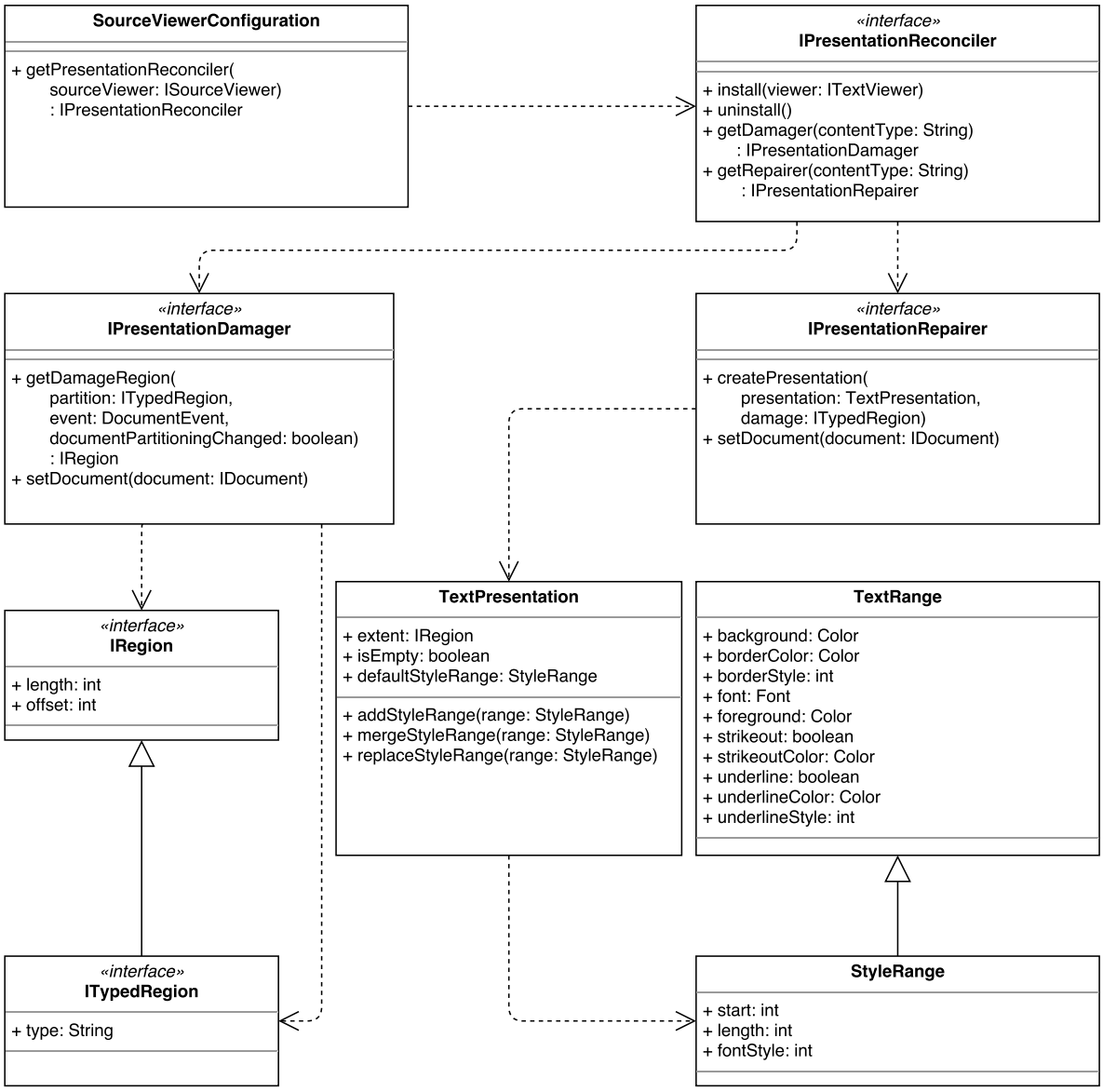


Figure 4.5: Overview of the API for the syntax coloring in Eclipse.

It is possible to register custom styles, but the `DefaultLanguageHighlighterColors` class provides the `TextAttributesKey` objects for pre-defined styles that change when the user selects a different color scheme.

Notepad++

Notepad++ uses the Scintilla editor component under the hood (Notepad++ 2009), which means that plugins that want to contribute functionality to Notepad++ have to use the Scintilla API. The most relevant parts of the API for syntax coloring are shown in Figure 4.7.

A Scintilla lexer object has to implement the the `ILexer4` interface with its `lex` method, which is given a document and a range within that document. The implementation should then call the `startStyling` and `setStyleFor` methods on the given `IDocument` to indicate the styling of ranges of source code. It is possible that a change in the current document invalidates more than the range the editor asked for, for example when the user has not closed a multi-line comment. When the lexer can determine this, it should call the `ChangeLexer-`

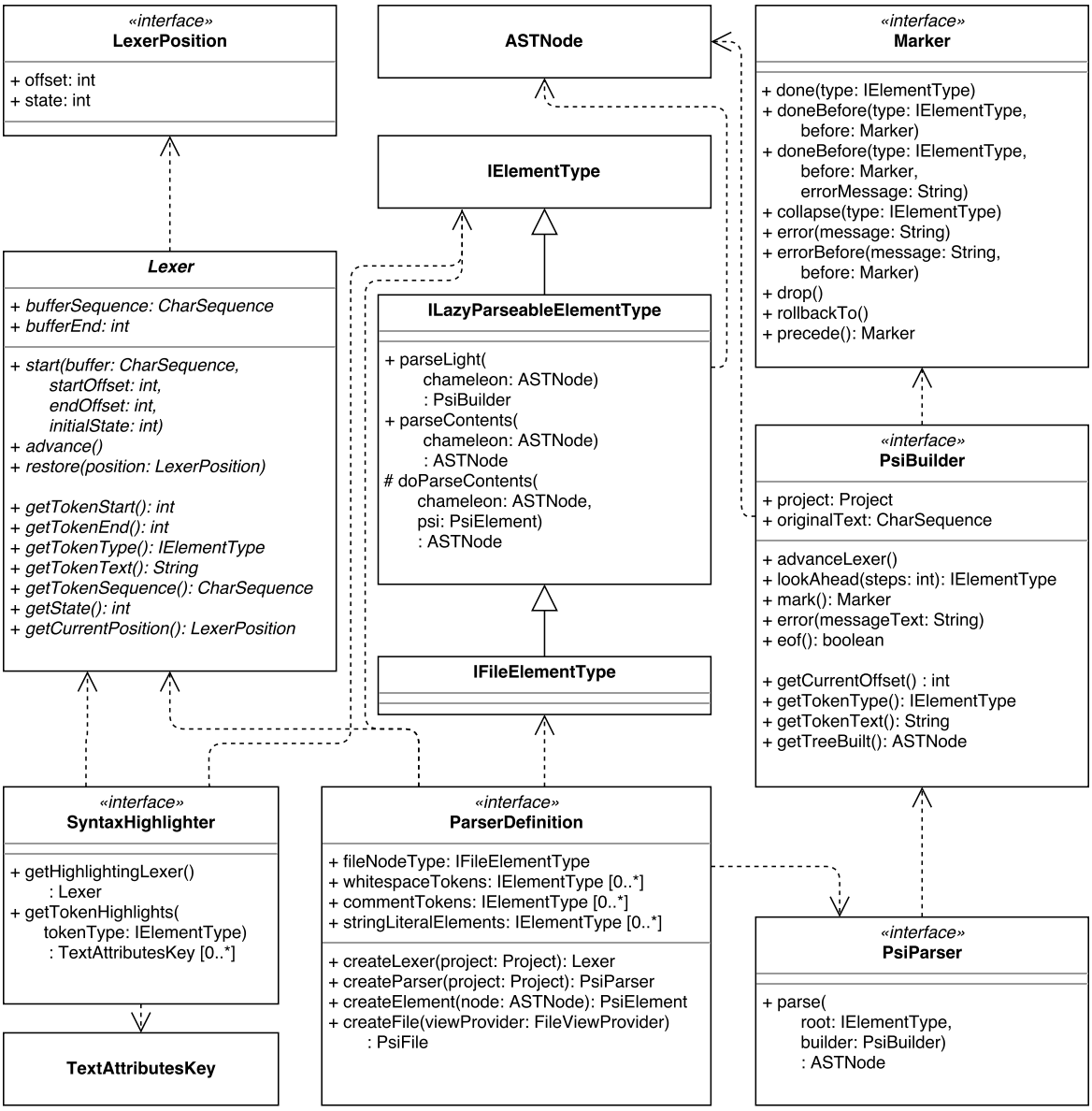


Figure 4.6: Overview of the API for the syntax coloring in IntelliJ.

`State` method on the document to indicate the portion of the document that needs to be restyled. (Scintilla 2018a)

The style of a range of text is indicated by a style's *index*. A style has a name, description, and list of tags. The tags are a space-separated list of identifiers, from most specific to least specific. For example, `error comment documentation keyword` could be the tags for a error in a documentation comment keyword. A color scheme assigns colors, styles, and fonts to individual tags and combinations of tags. (Scintilla 2018b)

The `IDocument` class has methods for getting the content and length of the document, and for converting between line-character offsets and absolute offset within the document. However, the name and path of the source file are not available to the lexer.

4.1.4 Comparison

Syntax coloring must be fast, as slow syntax coloring while the user types is distracting at best and confusing at worst. We assume this is the reason most editors support only a declarative syntax coloring specification. As shown in Table 4.8, most editors support TextMate gram-

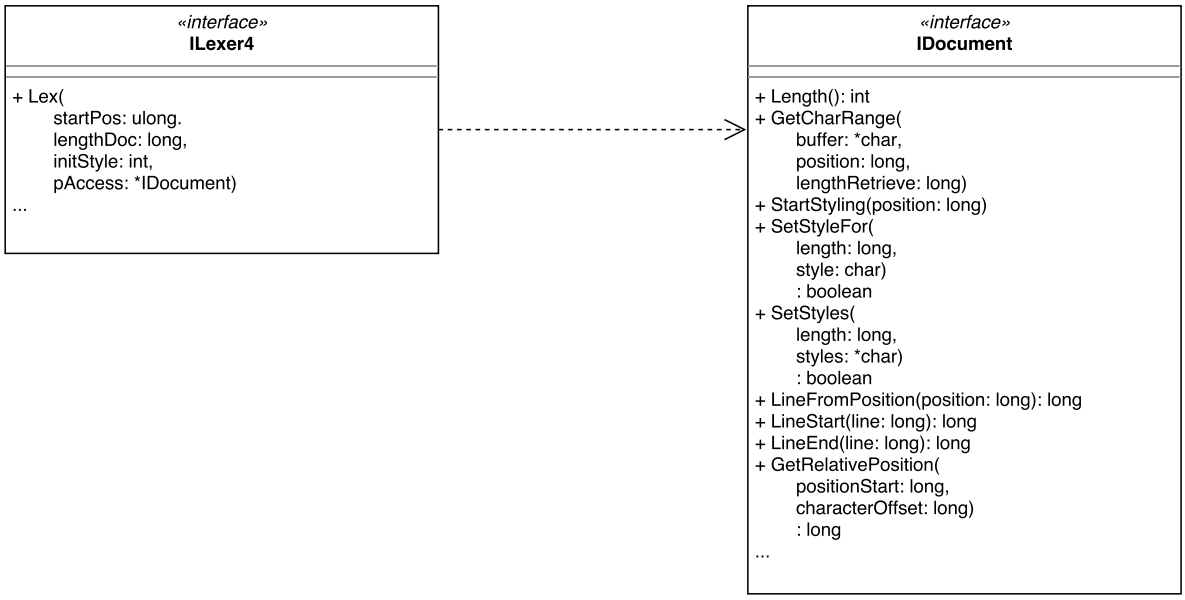


Figure 4.7: Overview of the API for the syntax coloring in Notepad++.

mars, and only Eclipse, IntelliJ, Visual Studio, and Notepad++ expose a syntax coloring API. All editors support color schemes, but Eclipse needs the Eclipse Color Themes[3] plugin for this.

	VS	NP++	Sublime	Vim	Eclipse	IntelliJ	VS Code	Atom	Cloud9	Che
Support										
TextMate	●	○	●	○	◒	◒	●	●	●	●
Custom syntax DSL	○	●	○	●	○	○	○	○	○	○
Color schemes	●	●	●	●	◒	●	●	●	●	●
API	●	●	○	○	●	●	○	○	○	○

Table 4.8: Comparison of syntax coloring support across editors.
(●, supported; ◒, supported through third-party plugins; ○, not supported)

[3] http://www.eclipsecolorthemes.org/

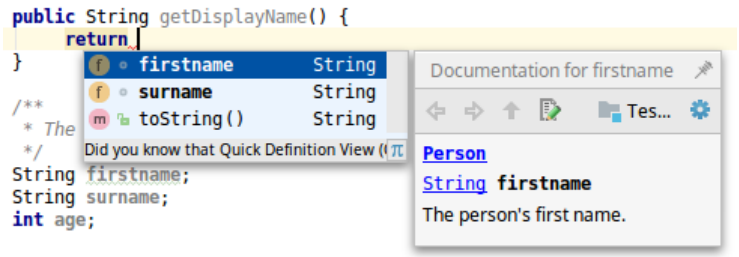


Figure 4.9: Smart code completion in IntelliJ shows only proposals that are compatible with the given type. For each proposal it shows their name, type, icons to indicate their kind and type, and some documentation for the selected candidate.

4.2 Code Completion

Code completion is the editor service that suggests syntactic constructs and identifiers for the user to quickly insert at the caret location without having to type them out completely. When triggered, the editor service displays a list of completion proposals for the user to choose from. Editors can show metadata along with each proposal, such as the (return) type of a declaration.

There are two kinds of code completion: syntactic, and semantic. Syntactic code completion suggests only syntactic language constructs, whereas semantic code completion suggests the names of symbols such as variables, fields, functions, and classes that are in scope at the caret location. The code editors we looked at make no distinction between syntactic and semantic code completion.

Code completion is triggered manually when the user presses a specific key combination, or automatically when the user types certain language-specific characters. For example, when the user types the member access operator *dot* (`.`) after an object's identifier in Java, code completion is triggered and lists the names of members of that object. To filter the list, the user can type the first few characters of the desired proposal.

The user can commit to a completion proposal by pressing a special key such as *enter* or *tab*. Upon committing to a proposal, its text is inserted into the document at the current caret location. Any prefix that is already in the document is also replaced. Some editors additionally support *snippets*, which are completions that have one or more placeholders for the user to fill out. The completion for an `if`-statement, for example, could have placeholders for the condition, then-branch, and else-branch.

In the next subsection we will compare code completion support across editors, in particular their API and its features and requirements. In Section 4.2.2 we provide an overview of the features each editors provides.

4.2.1 Editor Support

Of the editors we investigated only Notepad++'s support for code completion is not customizable, as it can only suggest words it has found in the current document. The other editors all have a dedicated API for code completion, but they vary in the features they support.

Visual Studio

Completion proposals in Visual Studio are represented by the `Completion` class, as shown in Figure 4.10, which has a label, description, and icon. The text to be inserted can be customized as well.

A code completion session in Visual Studio is triggered through a *command*, such as a key press, button click, or menu selection. There is only a general API for registering com-

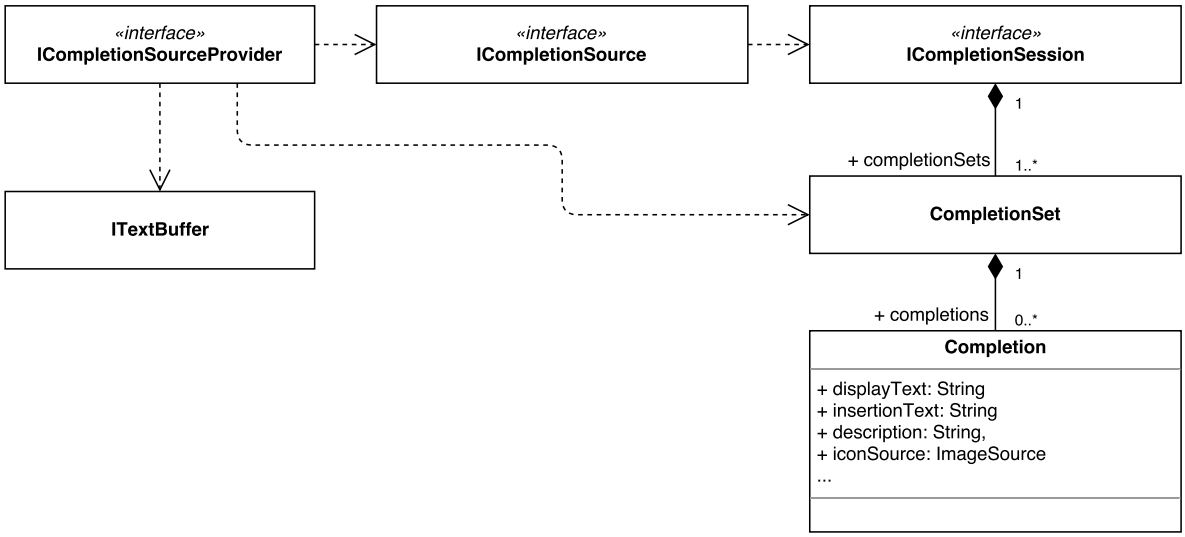


Figure 4.10: Overview of the API for code completion in Visual Studio.

mand handlers, which is also used for the commands that commit to a certain completion, or dismiss the completions altogether. The implementation does not get the current document or caret location, but can retrieve these from the `ITextBuffer` using which the completion source was created. (Microsoft 2016f)

Eclipse

The main class to implement for code completion in Eclipse is the *content assist processor*, as shown in Figure 4.11. When code completion is triggered — the processor can specify which characters trigger code completion — it is given the current document's text viewer and caret location, and should return a list of completion proposals. A proposal has a (styled) display name, optional description text, icon, and a list of characters that would cause the completion proposal to be accepted.

As the logic for accepting a completion proposal must be provided by the language implementation, it can range from simply inserting some text to providing code snippet support with placeholders. This additional flexibility can also be used, for example, to move the caret between the parenthesis of an inserted function call, or add the appropriate `import` statement at the top of the file when completing a type in Java. (Eclipse 2018c)

IntelliJ

IntelliJ has two types of code completion: basic, and smart. Basic code completion just suggests the code fragments and identifiers for the current caret location. However, smart code completion tends to be a little smarter, and only suggests those completions that actually make sense at the insertion point, based on for example the type of the expression.

The user can invoke code completion more than once, which displays different results each time. For example, the first time Java code completion is triggered, it will show local variables and fields visible from the current caret location. Triggering code completion a second time will also show static fields and methods, and any classes in dependencies. Code completion can be triggered a third and final time to display all classes, both in the project and its dependencies.

There are a few ways in which code completion can be implemented. When a PSI structure is available, code completion can be implemented directly on a PSI reference or in a

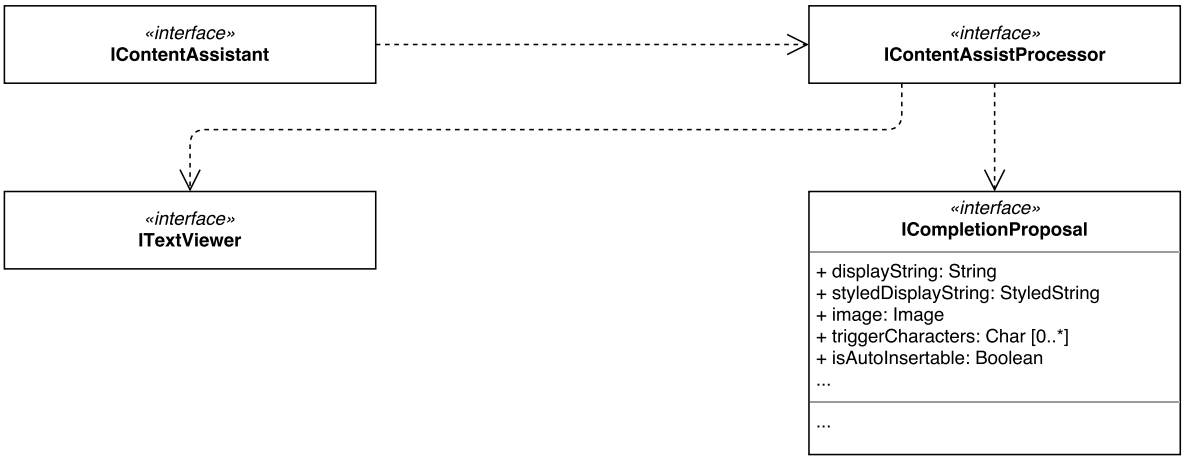


Figure 4.11: Overview of the API for code completion in Eclipse.

completion provider. Otherwise, the most flexible approach is to implement code completion in a *completion contributor*. The relevant API is shown in Figure 4.12.

However code completion is implemented, the completion proposals can be represented as either PSI elements, or *lookup elements*. For PSI elements the editor extracts the relevant metadata, such as the label and icon. Otherwise, for lookup elements the implementation gets to determine the metadata. IntelliJ's built-in `LookupElementBuilder` supports building lookup elements with a styled label, icon, type text, and tail text. Tail text is text that is shown directly after the label, such as function parentheses, whereas type text, usually the (return) type of the declaration, is shown at the right-hand margin. (JetBrains 2018c; JetBrains 2018p)

VS Code

In VS Code, which uses Language Server Protocol (LSP), code completion is implemented as part of a language server. Given the document, caret location, and information on what triggered the code completion, the service returns a list of completion proposals. The list may be considered incomplete, forcing the editor to call upon code completion to recompute the list of proposals when the user types some more characters.

The list consists of completion proposals, each having at least a label. A simple proposal just inserts the label's text when the user commits to it, but more complex proposals can specify the text edits to apply, or execute arbitrary complex logic through a *command*. The icon shown next to the proposal in the editor is one of a pre-defined list of kinds. The relevant API can be seen in Figure 4.13. (Microsoft 2018e)

Sublime

The code completion service API of Sublime, shown in Figure 4.14, is simple to implement but supports a few more advanced features such as multi-caret and snippet support. In Sublime a user can type with more than one caret at the same time, and therefore code completion also supports multiple carets.

Each completion proposal returned by the implementation has a label (called *trigger*), and the text content to insert. The label is also used for filtering the list of results. A hint value, displayed in gray at the right-hand margin of the completions list, can be specified by appending it to the label string. The text content can be a code snippet, with placeholders for the user to provide. (Sublime Text 2018a)

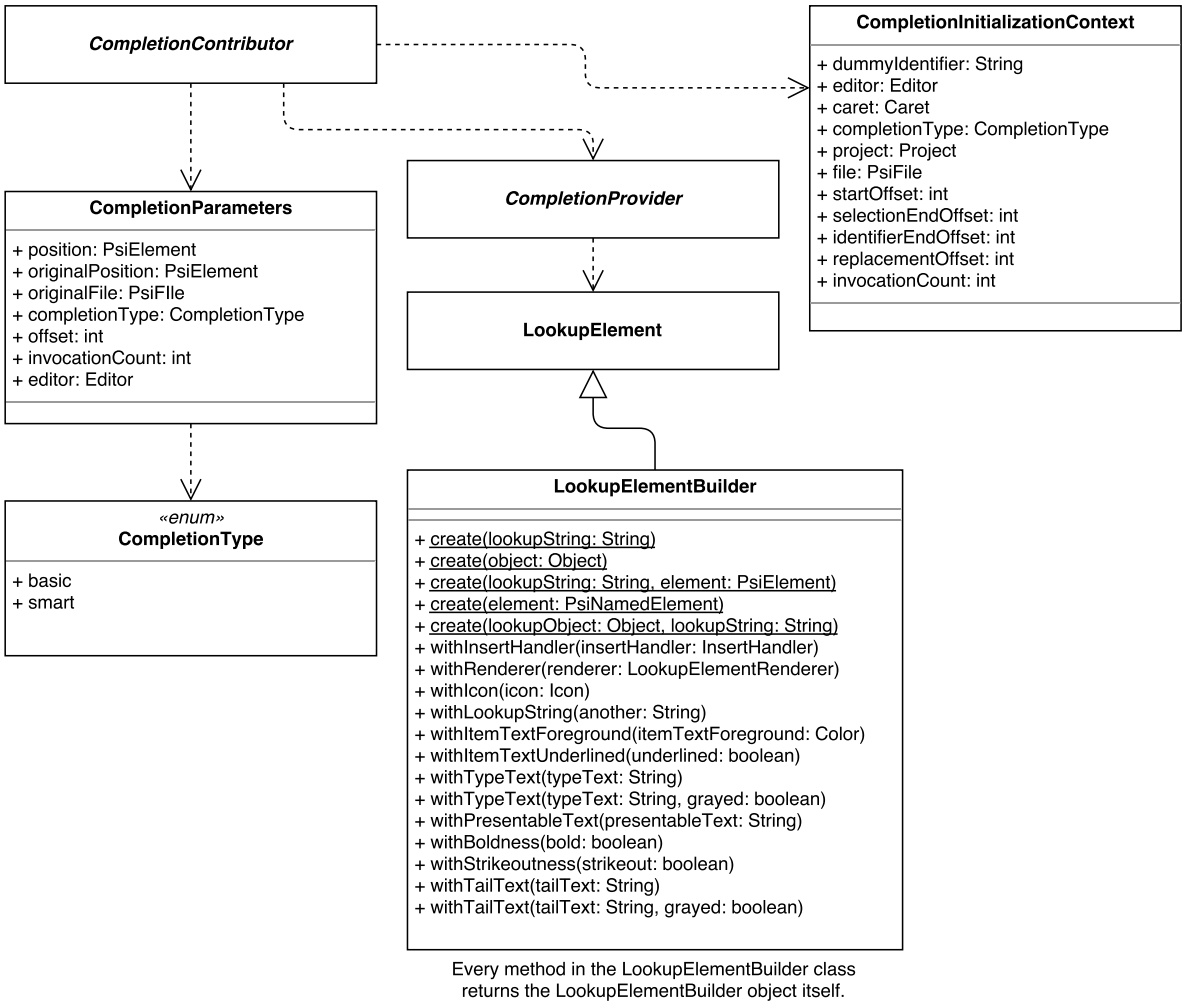


Figure 4.12: Overview of the API for code completion in IntelliJ.

Atom

Atom has no dedicated API for code completion, but there are third-party plugins that add code completion, such as the Autocomplete+ plugin. This plugin, which is the most popular code completion plugin for Atom, exposes an API for code completion, which is shown in Figure 4.15.

Each completion proposal is represented by a `Suggestion` object, for which the text to insert is the only required field. This is also the text displayed to the user unless overridden by providing a display text. Settings the *replacement prefix* overrides the text to be replaced. The icon can either be one of the predefined values, or a custom icon. The left and right labels, or their HTML counterparts, are shown before and after the suggestion's text respectively. These can be used to show, for example, the return type and arguments of a function. The description, if present, is shown for the selected suggestion, followed by a link to more documentation when `descriptionMoreURL` is specified.

Vim

A language plugin in Vim can implement a completion function to be invoked when completions are triggered. This function returns a list of completion proposals that include the

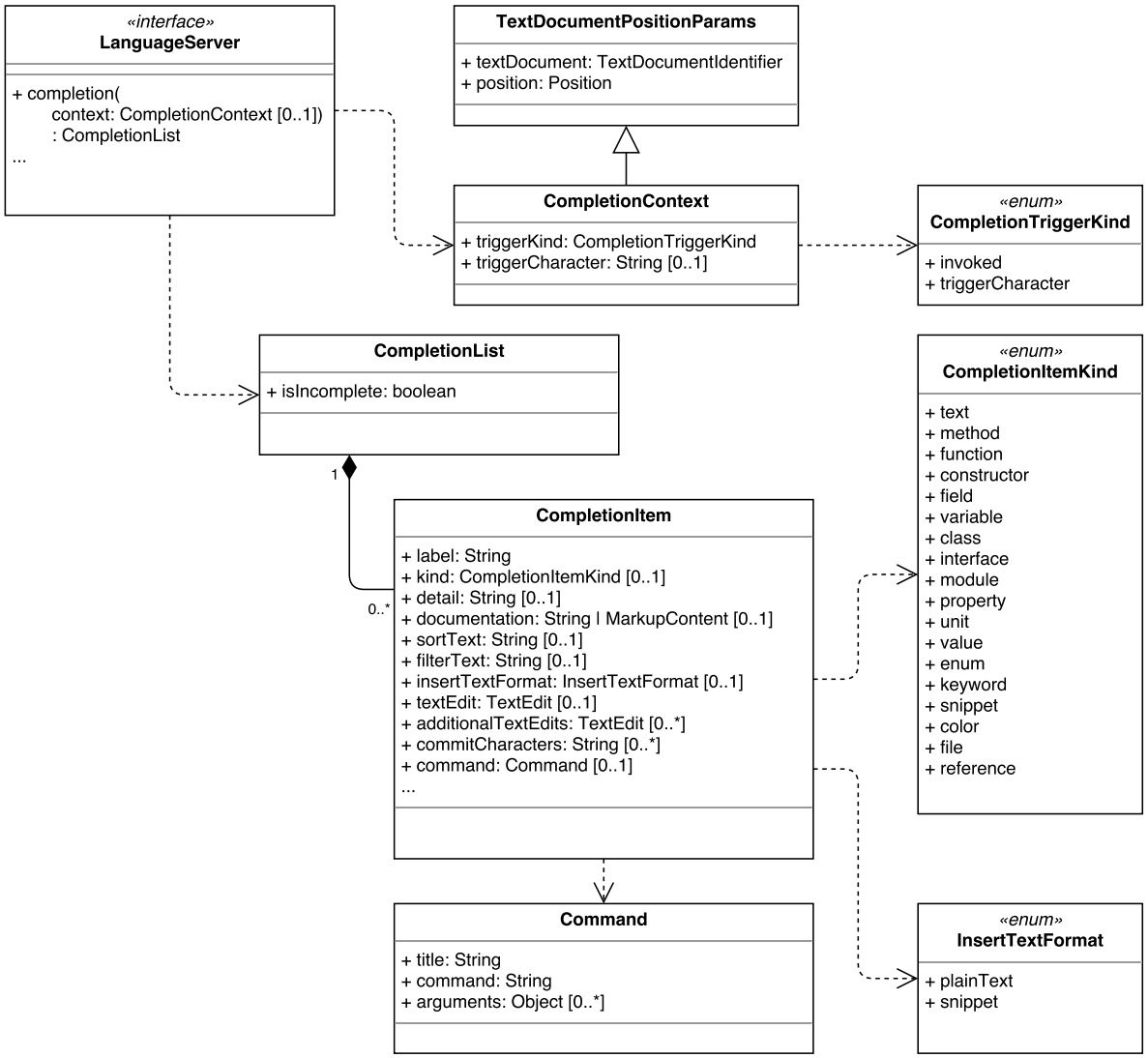


Figure 4.13: Overview of the API for code completion in LSP.

Figure 4.14: Overview of the API for code completion in Sublime.

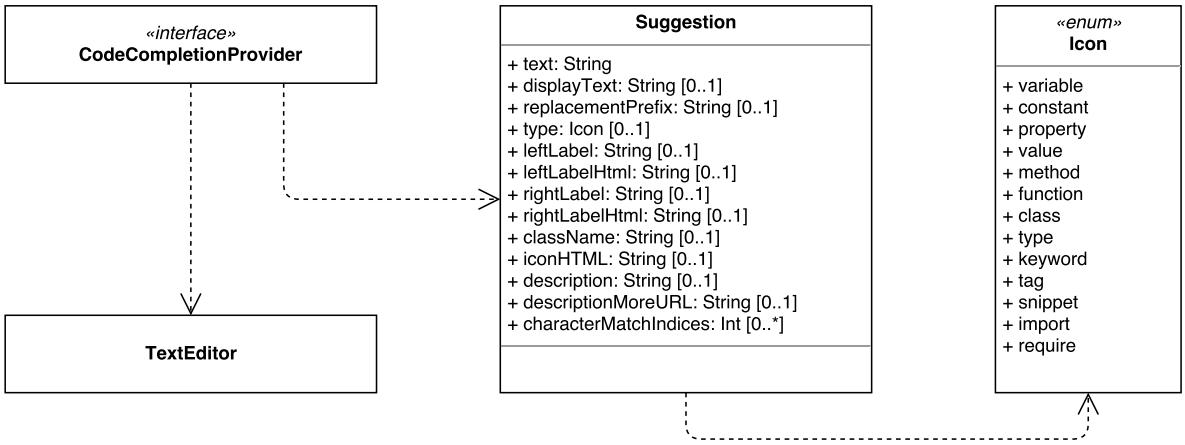


Figure 4.15: Overview of the API for code completion in the Autocomplete+ plugin for Atom.

Figure 4.16: Overview of the API for code completion in Vim.

display name, text to insert, extra text to display after the display name, a description, and the kind of completion. The relevant API is shown in Figure 4.16. (Vim 2017)

Cloud9

To implement custom code completion in Cloud9, the language plugin has to override the `complete` function shown in Figure 4.17. The function gets the current document, AST, and caret location within the document, and returns a list of completion proposals through a callback function. Each proposal includes a display name, one of the predefined icons, documentation, the text to insert, and meta-text, which is displayed in gray on the right-hand margin. (Cloud9 2018d)

While Cloud9 supports snippets, they can only be defined as files in a plugin bundle. Code completion has no dedicated API for inserting snippets.

Eclipse Che

In Eclipse Che code completion can be implemented either as part of a language server, or using the editor's own API. The LSP language server API for code completion is shown in Figure 4.13, whereas Eclipse Che's own API is shown in Figure 4.18. For the latter, a language plugin must implement a *code assist processor* that for a given document and offset calls a callback with a list of completion proposals. By default completion proposals include a display name and an icon, and can programmatically apply the completion. (Eclipse Che 2017a)

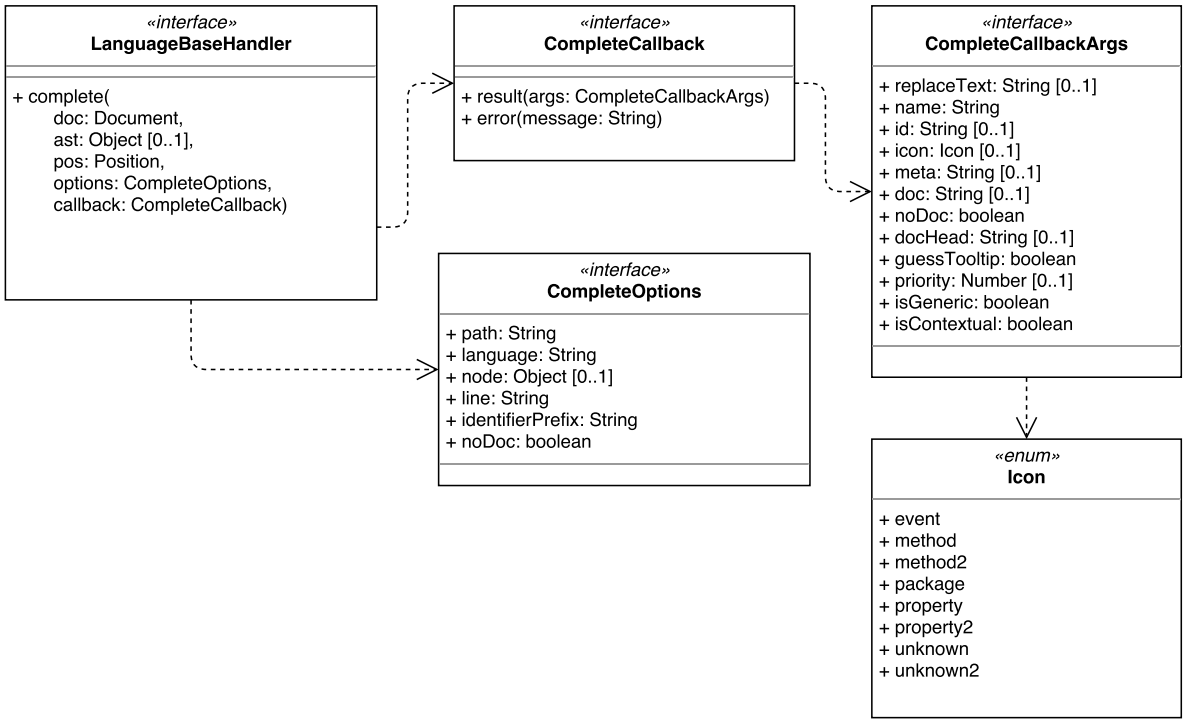


Figure 4.17: Overview of the API for code completion in Cloud9.

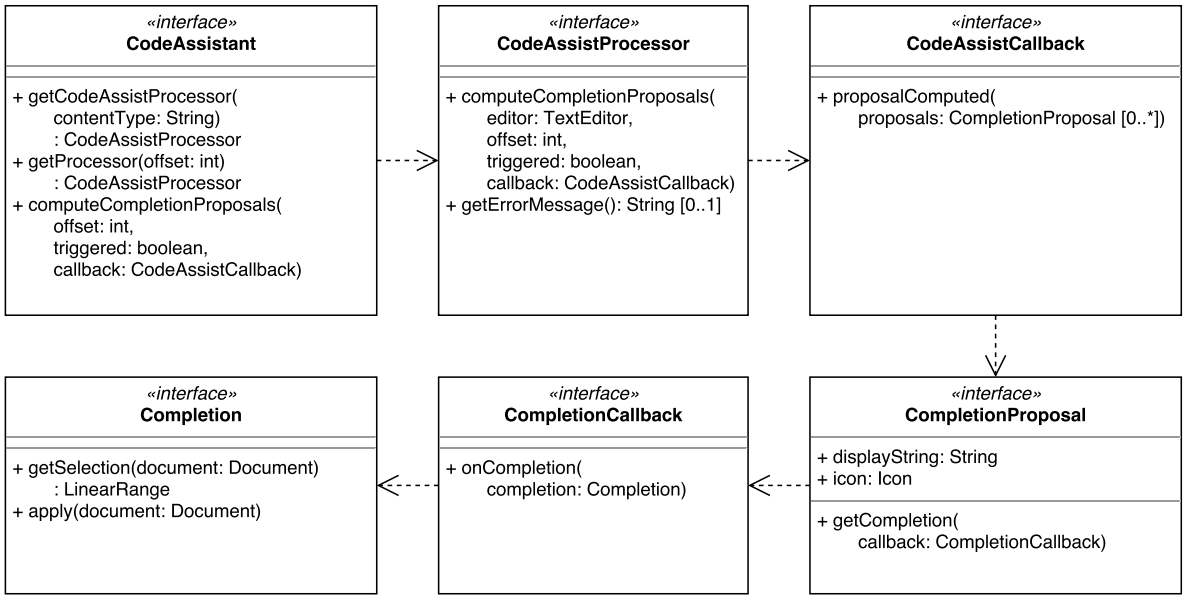


Figure 4.18: Overview of the API for code completion in Eclipse Che.

4.2.2 Feature Comparison

All editors except Notepad++ support simple completion proposals consisting of a label to be displayed to the user and the text to insert at the caret location. Most editors support code snippets, a description, and an icon, but often only an icon from a predefined list. Three of the editors can show documentation for the selected proposal, and text to the right of the proposal (such as the type of an identifier). The remaining features, including styled texts, are only supported by one or two editors. The full comparison is shown in Table 4.19.

	VS	Sublime	Vim	Eclipse	IntelliJ	VS Code	Atom	Cloud9	Che
Completion Proposal									
Label	●	●	●	●	●	●	●	●	●
Label (styled)	○	○	○	●	●	○	○	○	○
Description	●	○	○	●	○	●	●	○	○
Icon	●	○	◒	●	●	◒	●	◒	◒
Left text	○	○	○	○	○	○	●	○	○
Left text (styled)	○	○	○	○	○	○	●	○	○
Right text	○	●	●	○	●	○	●	●	○
Right text (styled)	○	○	○	○	○	○	●	○	○
Documentation	○	○	●	○	○	●	○	●	○
Styled Documentation	○	○	○	○	○	●	○	○	○
'More info'-URL	○	○	○	○	○	○	●	○	○
Custom sort text	○	○	○	○	○	●	○	○	○
Custom filter text	○	○	○	○	●	●	○	○	○
Triggering									
Custom trigger characters	●	○	○	●	○	○	○	○	○
Smart code completion	○	○	○	○	●	○	○	○	○
Multiple invocations	○	○	○	○	●	○	○	○	○
Committing									
Snippets	○	●	○	●	○	●	●	○	●
Multiple carets	○	●	○	○	○	○	○	○	○
Custom replacement text	●	●	●	○	●	●	●	●	○
Custom replacement logic	○	○	○	●	●	●	●	○	●
Custom commit characters	●	○	○	●	○	●	○	○	○

Table 4.19: Comparison of code completion editor service features across editors with a dedicated API.
(●, customizable through API; ◒, pre-defined choice; ○, no API)

4.3 Structure Outline

The structure outline, also known as the structure tree, outline tree, or outline view, is a tree representation of declarations in a file, such as classes, methods, and fields. The structure outline shows a high-level overview of the structure of the current file and allows users to quickly navigate to a declaration. The kind of declaration is often indicated by an icon, and some editors can also show the (return) type of the declaration, as shown in Figure 4.20.

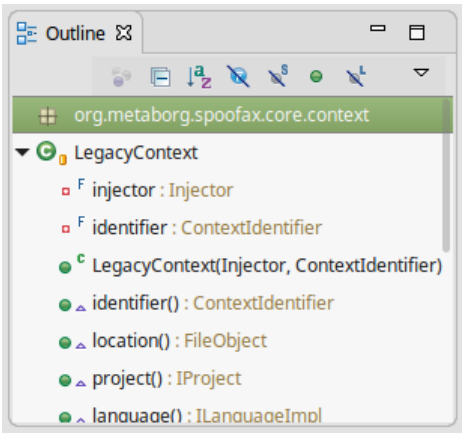


Figure 4.20: Structure outline in Eclipse showing the declarations in the current file as a tree. Each declaration has a type annotation and an icon indicating their kind.

4.3.1 Editor Support

Curiously, most editors we looked at do not support the structure outline directly. For Vim, Atom, and Sublime there are third-party plugins that add a structure outline based on Ctags or a similar declaration (Larres 2017; xndcn 2018; TextMate 2018). VS Code does not support a structure outline view yet, and can show only a flat list of symbols declared in the current file. (Microsoft 2016k) While Visual Studio has a project-wide structure outline in its *class view*, only Cloud9, IntelliJ, and Eclipse support a per-file structure outline tree. We will compare the features of these editors.

Eclipse

The nodes in a structure outline in Eclipse can be represented by a tree of arbitrary objects, each with a label and icon. Figure 4.21 shows the API. For a given node object, a `ITreeContentProvider` interface provides a list of child objects and a `ILabelProvider` provides the text and icon. (Eclipse 2018d)

There is no dedicated API for adding type information or other meta-data to the tree elements, but the tree node labels can be decorated as shown in Figure 4.20 using a special `DecoratingLabelProvider`. (Eclipse 2018f)

IntelliJ

IntelliJ leverages its PSI structure for the structure outline. A language has to select a subset of `PsiElement`-derived classes which are shown as tree elements in the structure outline. These elements may be extracted from the source code, or created on-the-fly. The PSI elements themselves define their label, icon, and general appearance such as whether to show a plus button to extend the element. They also return a list of child elements to show in the structure outline. The correspondence between the code and the structure PSI allows the user to navigate from the structure outline to the code and vice versa. (JetBrains 2018n)

The API in Figure 4.22 shows that a structure outline node, represented by `PsiTreeElementBase`, has a label, icon, and list of children. The node can also have a *location string*, which is a location label such as the package of a class, usually displayed next to the item name in a different color. The `navigate` method is called to navigate to the source code when the user clicks the node. The tree elements are part of the `StructureViewModel`, created by a

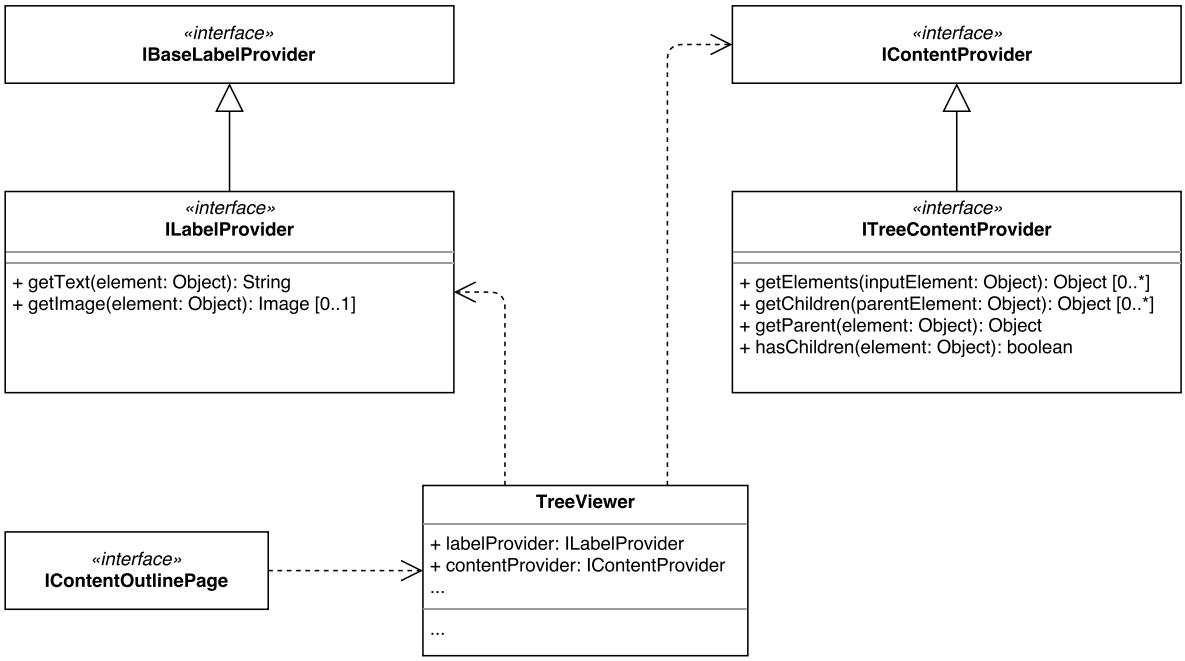


Figure 4.21: Overview of the API for the structure outline in Eclipse.

`TreeBasedStructureViewBuilder` that is called by an implementation of the `PsiStructureViewFactory`.

Cloud9

A source file's structure outline in Cloud9 shows a tree of outline items, each with a name, icon, range of the full structure in the source code, and range of the name of the structure in the source code.

The main method to implement is the `outline` method of the `LanguageBaseHandler`, as shown in Figure 4.23. It is given the current document, and possibly an AST if the language provided any, and should call the callback's `result` function with the root `OutlineItem` of the structure outline tree. The icon of an outline item must be one of the predefined values. (Cloud9 2018d)

4.3.2 Feature Comparison

The editors Eclipse, IntelliJ, and Cloud9 that support a file-based structure outline editor service have similar features, as shown in Table 4.24. All allow the tree nodes to have a label and an icon, albeit Cloud9 only allows a pre-defined list of icons, and can navigate to the corresponding source code. While IntelliJ has special support for specifying a location label on a node, Eclipse node labels can have arbitrary extra information by using a label decorator.

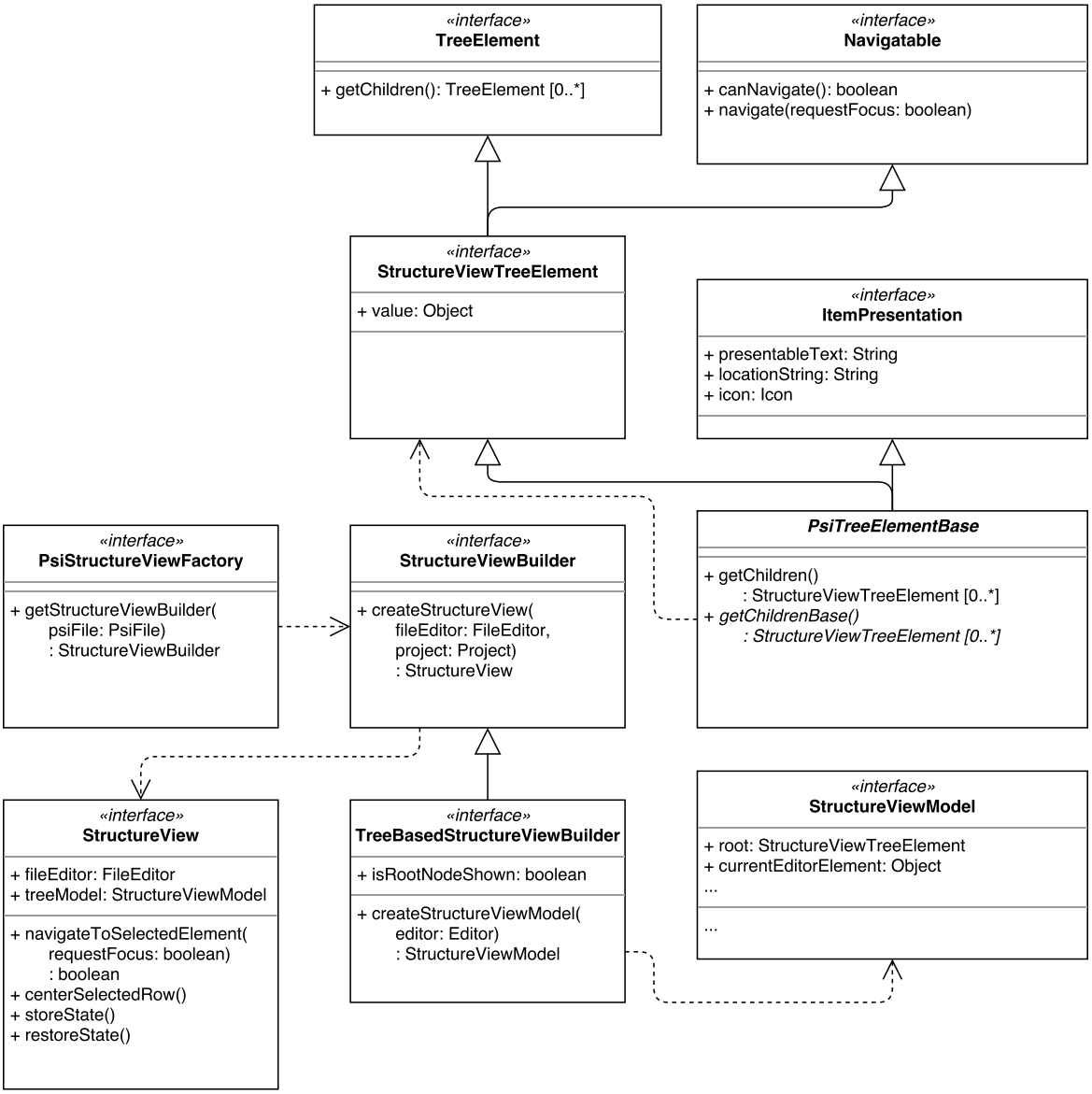


Figure 4.22: Overview of the API for the structure outline in IntelliJ.

	Eclipse	IntelliJ	Cloud9
Tree Nodes			
Label	●	●	●
Styled label	●	○	○
Icon	●	●	◒
Location label	○	●	○
Navigation			
Navigation	●	●	●
Navigation event	●	●	○

Table 4.24: Comparison of structure outline editor service features across editors with a dedicated API.
(●, customizable through API; ◒, pre-defined choice; ○, no API)

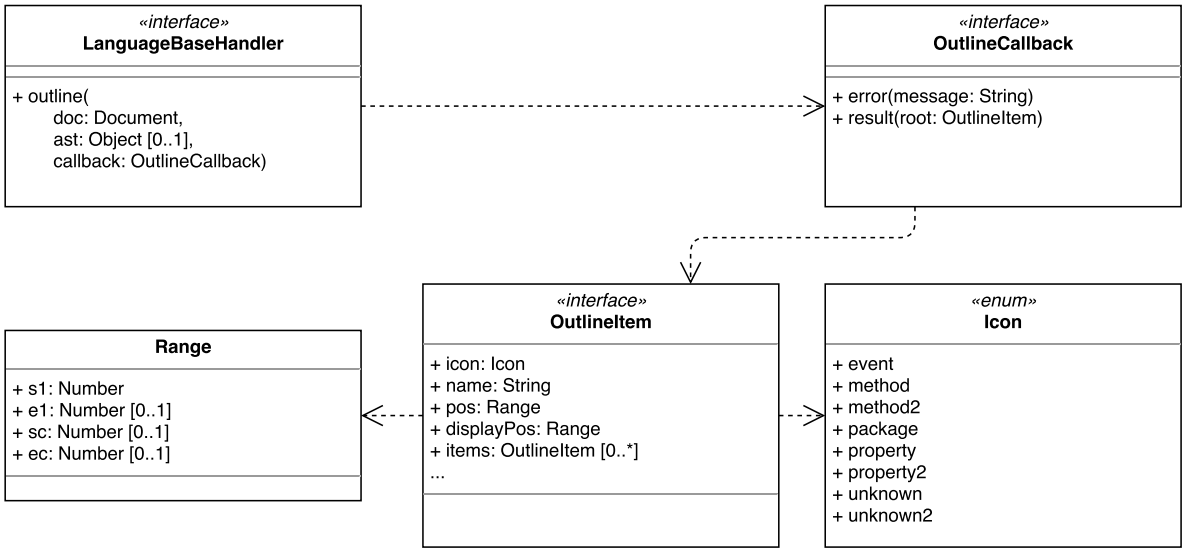


Figure 4.23: Overview of the API for the structure outline in Cloud9.

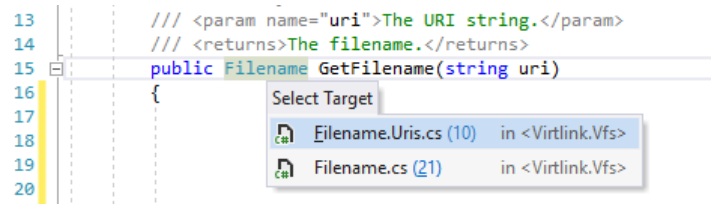


Figure 4.25: Reference resolution in Visual Studio asking the user which of the two definitions of `Filename` they want to go to.

4.4 Reference Resolution

Most editors we looked at support reference resolution, which allows the user to instantly jump to the definition of the symbol under the caret, or, when there is more than one definition, list the definitions and let the user pick one to jump to, as shown in Figure 4.25. The latter happens, for example, with `partial` classes in C♯ that spread the definition of a class across several files.

4.4.1 Editor Support

Those editors that have programmatic support for reference resolution usually have a dedicated API, with the exception of Visual Studio and Eclipse. Simpler editors instead use a static list of symbols for navigation, such as *Ctags*.

Ctags

Ctags is a format for symbol indices, listing the symbols and their locations in the various files of the current project. Ctags indices are generated ahead-of-time and used to provide symbol navigation in code editors such as Vim, and through plugins in Notepad++ and Atom.

Various tools can generate Ctags indices, such as the language-agnostic Exuberant Ctags and Universal Ctags tools, or language-specific tools such as Hasktags for Haskell and Jsctags for JavaScript. With many tools capable of generating Ctags for various languages, the format of the Ctags indices is not really standardized. However, the most popular tools generate very similar indices that can be read by most editors that support Ctags. An entry in the index specifies at minimum the name, file, and location of the symbol. Optional fields include the symbol's kind, visibility, and the signature if it represents a function. (Hiebert 2017)

TextMate Symbol List

Sublime uses the TextMate symbol list system for its *Go to Definition*, *Show local symbol list* and *Show global symbol list* features. Symbols are stored in the local (file-based) or global (project-wide) symbol index, and are discovered by their TextMate grammar scope names (see Section 4.1.1). If necessary, a substring regular expression match can be specified to get just the symbol's name. (Sublime Text 2018c)

IntelliJ

For this editor service IntelliJ again leverages its PSI structure. PSI elements that implement the `getReference` and `getReferences` methods return their PSI reference objects. These reference objects have a `resolve` method which returns the PSI elements of the declarations referenced by the element. This will automatically enable *Quick Definition Lookup* and *Go to Declaration* in the editor. (JetBrains 2018j)

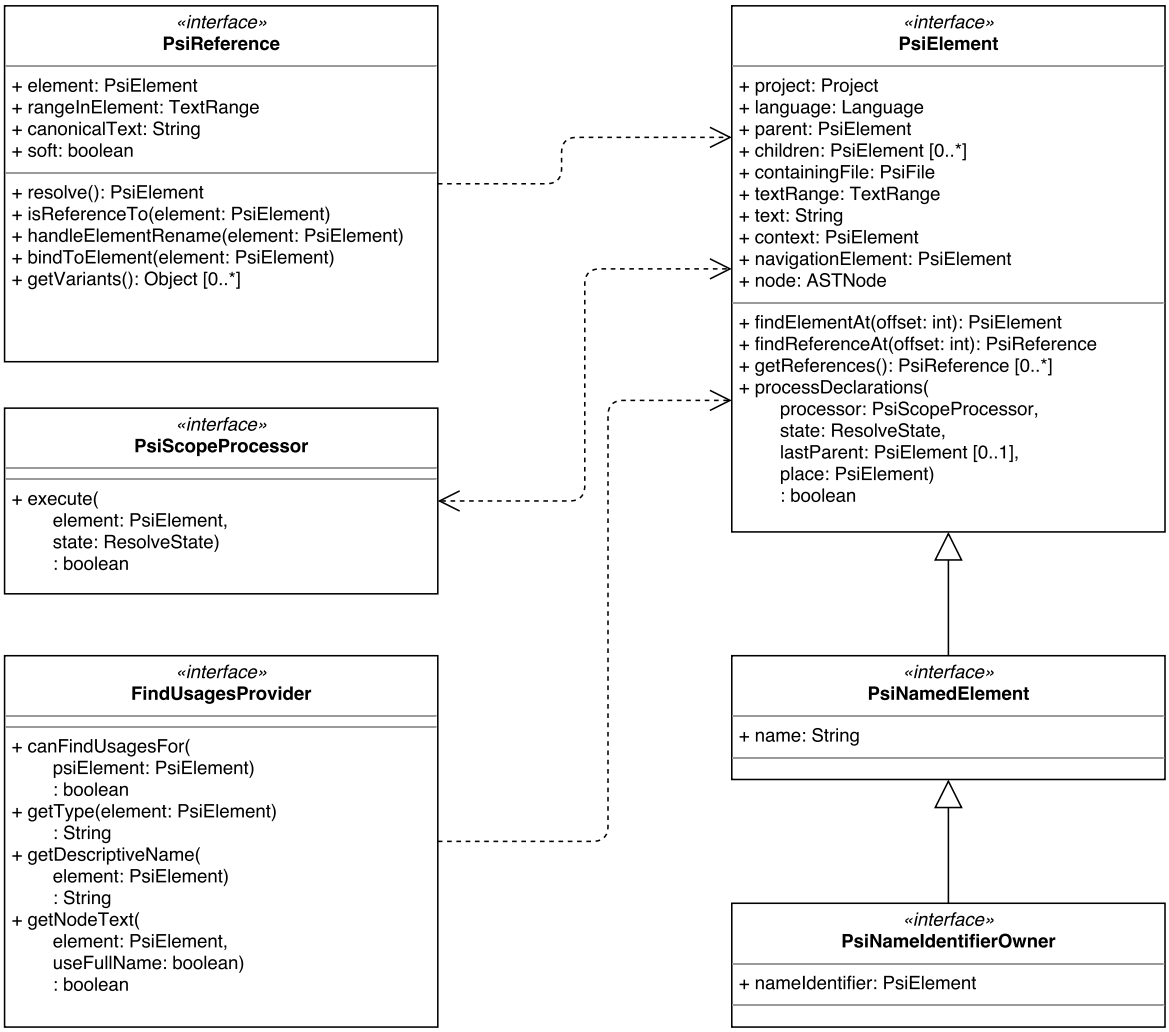


Figure 4.26: Overview of the API for reference resolution in Eclipse.

The *Find Usages* command in IntelliJ uses an implementation of the `FindUsagesProvider` interface. This implementation determines whether the action is available for the PSI element under the caret, and how to display the selected element to the user. IntelliJ maintains an index of words in every file (determined by the find usages provider), and uses that index to determine which files may contain the word to be found. IntelliJ then builds a PSI tree for each file. When the word to be found is an identifier, only the candidates that reference the identifier are returned. Otherwise, all literals that match the word are returned. (JetBrains 2018f) The API is shown in Figure 4.26.

VS Code

VS Code can show the user a preview of a symbol's definition without actually moving the caret and selection there, allowing the user to *peek* at the definition. In addition to navigating to a symbol's definition, VS Code supports navigating to a symbol's type definition or implementation. These editor services can be supported by implementing the `DefinitionProvider`, `TypeDefinitionProvider`, and `ImplementationProvider` respectively, as seen in Figure 4.27. Each provider takes the active document and current caret location, and is expected to resolve the symbol under the caret to a collection of locations. Each location represents a definition and indicates the document and range where the definition can be found. (Microsoft 2018c)

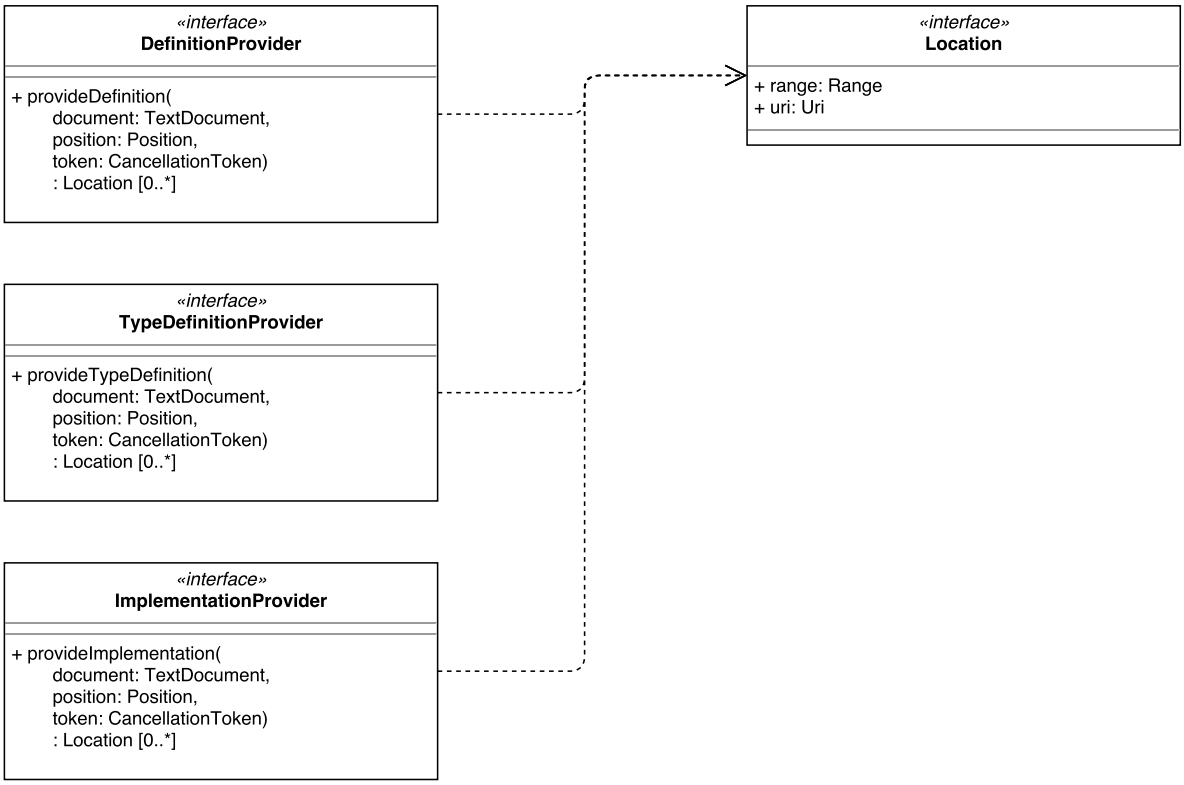


Figure 4.27: Overview of the API for reference resolution in VS Code.

Cloud9 and LSP

Reference resolution in Cloud9 and LSP (VS Code and Eclipse Che) is very similar and takes the current document and caret location and returns zero or more definitions. When there are multiple results, Cloud9 displays an icon next to each definition. (Cloud9 2018d; Microsoft 2018a) The Cloud9 API is shown in Figure 4.28.

Visual Studio

Visual Studio does not have a dedicated API for implementing reference resolution. Most languages implement reference resolution by hooking into the command API of Visual Studio, responding to the `VSStd97CmdID.GotoDefn` and `VSStd97CmdID.FindReferences` commands, for which they can execute a custom action.

Implementations can get the current caret location in the source text view, and use that to determine where to resolve to. Visual Studio can then open that file and move the caret to the name of the reference or definition. If there is more than one reference or definition, Visual Studio asks the user to select one of multiple definitions. (Microsoft 2016h)

Eclipse

As Eclipse does not have a dedicated API for reference resolution, it can be implemented by implementing a hyperlink detector, which highlights an identifier when the user hovers over it. Then when the user clicks the identifier, the caret is moved to the document and location of the definition.

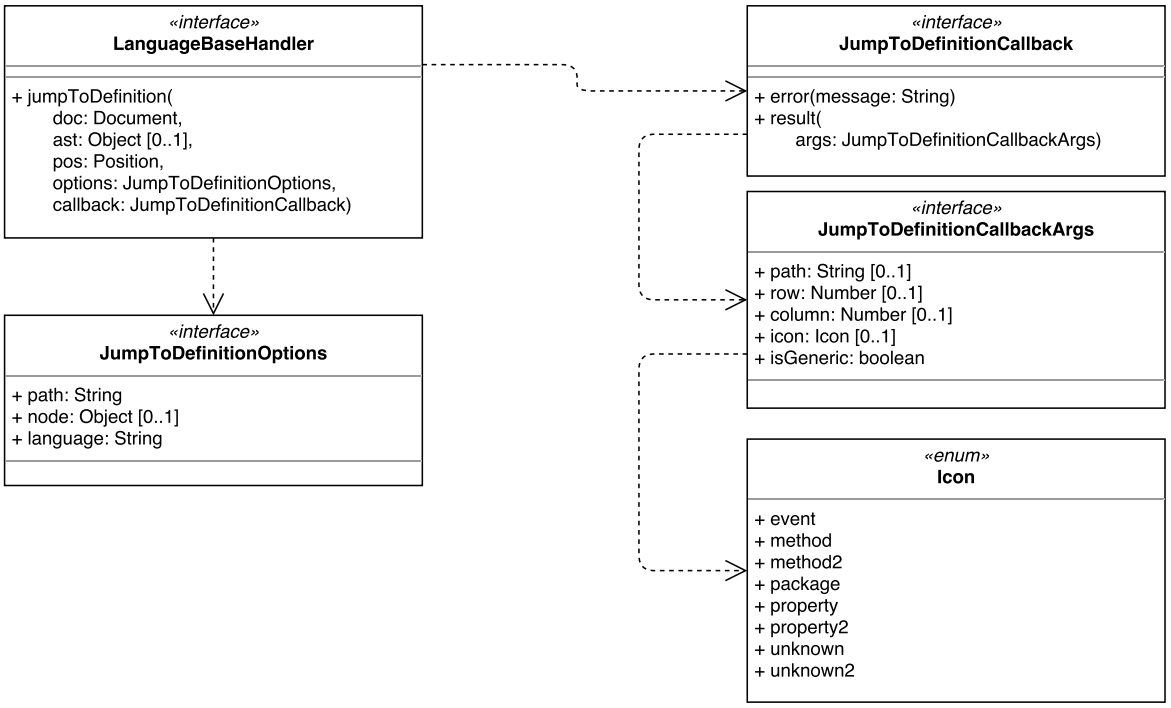


Figure 4.28: Overview of the API for reference resolution in Cloud9.

4.4.2 Feature Comparison

The APIs and features for reference resolution are very similar in the editors that support it. Only the Cloud9 API additionally supports an icon for the various definitions, which is displayed when the user is asked to pick one of multiple possible definition sites, as shown in Table 4.29.

	VS	Eclipse	IntelliJ	VS Code	Cloud9	Che
Resolve						
Definition	◒	◒	●	●	●	●
Multiple definitions	◒	○	●	●	●	●
Definition icon	○	○	○	○	●	○

Table 4.29: Comparison of reference resolution editor service features across editors with programmable support.
(●, dedicated API; ◒, general API; ○, not supported)

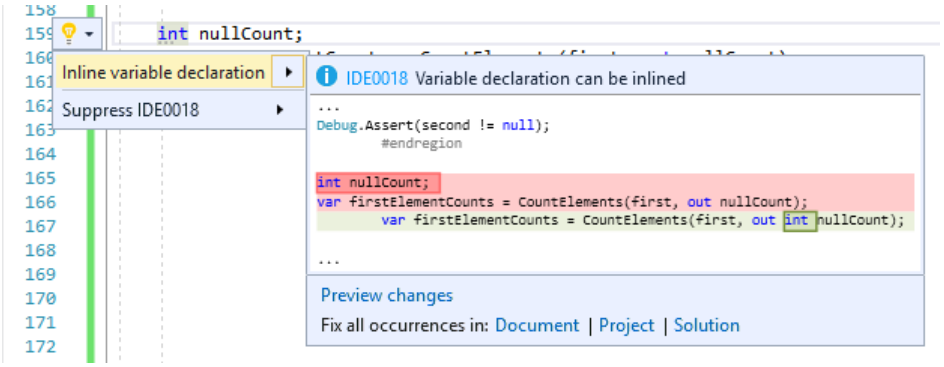


Figure 4.30: Code actions in Visual Studio show possible actions the user can perform on the code at the caret location, and can even give a preview of the result of the action on the code.

4.5 Code Actions

Code analyzers can identify issues, optimizations, and suggestions that can be applied to the user's code at the current caret location, and display these through *code actions*. Also known as *intention actions*, *quick actions*, *quick fixes*, and *quick assists*, the availability of code actions is usually signified by a light bulb icon in the editor margin. Activating this light bulb shows a list of possible refactorings, and when the user selects one the refactoring is automatically applied to their code.

4.5.1 Editor Support

Code actions are only supported by the more IDE-like editors we looked at, namely Visual Studio, Eclipse, IntelliJ, VS Code, and Eclipse Che. Code actions and refactorings are not supported by Sublime, Atom, Notepad++, or Vim. While Vim has a feature called *quick fixes*, it is simply a list of errors.

Visual Studio

The code actions in Visual Studio, as shown in Figure 4.30, are more advanced compared to the other editors, as Visual Studio code actions can also show a preview of the code after applying the refactoring.

The Visual Studio API for code actions is shown in Figure 4.31, where an implementation of a *suggested actions source* returns a list of suggested actions for the given document and caret location. An action suggestion has a display name, icon, and optionally a user interface (UI) object that provides a preview of the result of applying the action. (Microsoft 2016c)

Eclipse

Eclipse can provide code actions for parts of the source code that have been marked with an *audit marker*. As the API in Figure 4.32 shows, a *resolution generator* is asked which *resolutions* the specified marker has, if any, and each resolution provides an icon, label, and description. When the user chooses a resolution, its `run` method is invoked, which is given the marker on which the resolution was invoked. (Eclipse 2018a)

IntelliJ

A code action in IntelliJ is an implementation of the `BaseIntentionAction` class, shown in Figure 4.33, and is usually registered for a specific annotation. The actions are provided by an *annotator* implementation. (JetBrains 2018i)

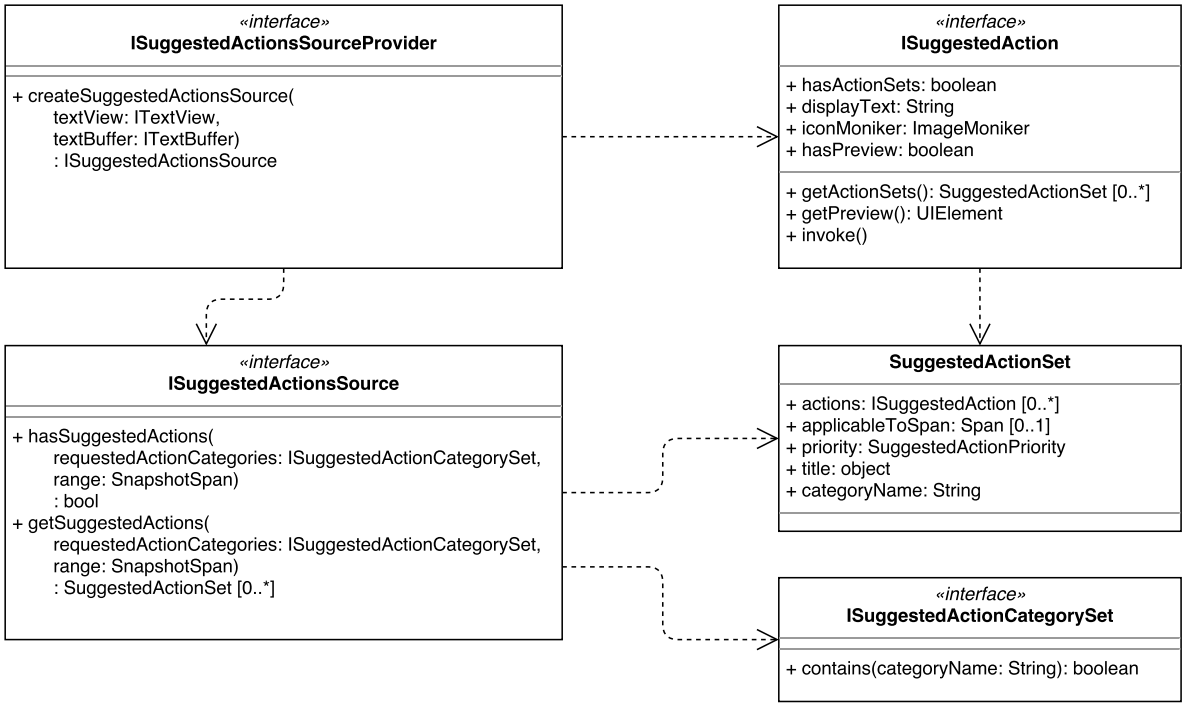


Figure 4.31: Overview of the API for code actions in Visual Studio.

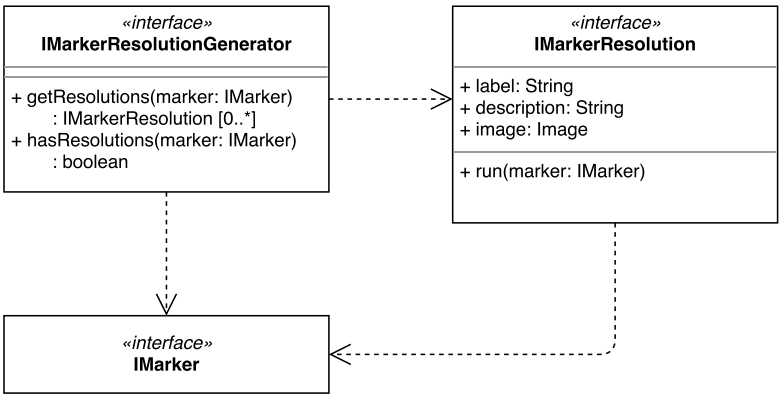


Figure 4.32: Overview of the API for code actions in Eclipse.

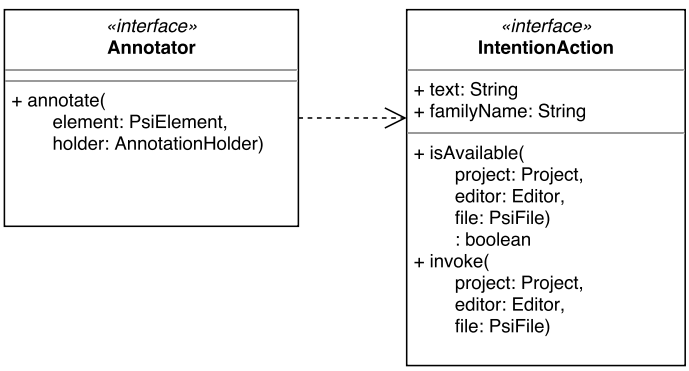


Figure 4.33: Overview of the API for code actions in IntelliJ.

Figure 4.34: Overview of the API for code actions in VS Code.

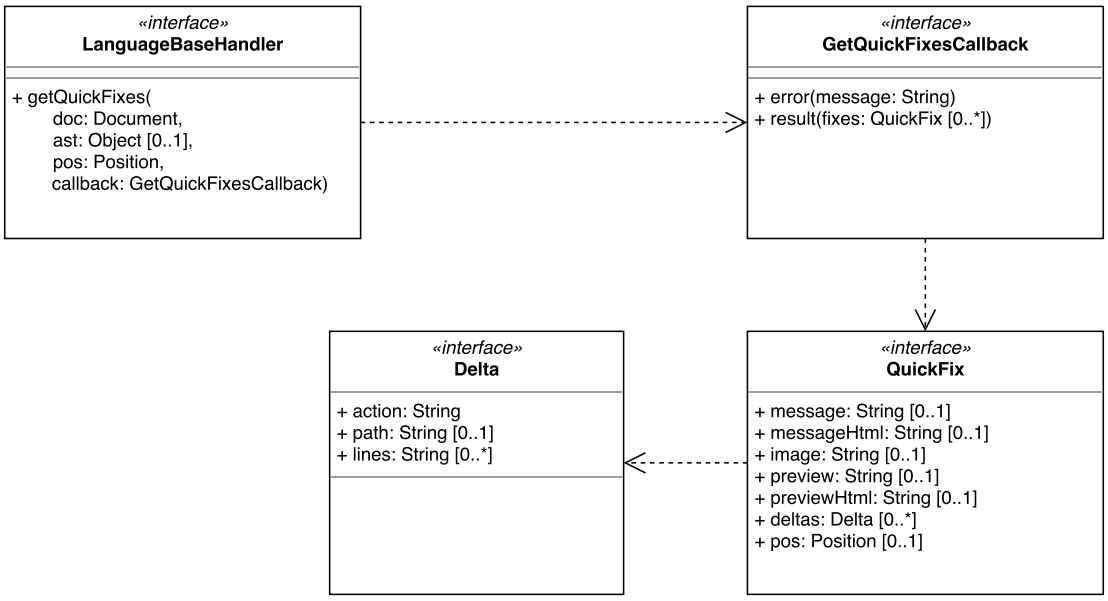


Figure 4.35: Overview of the API for code actions in Cloud9.

VS Code

Code actions in VS Code are handled by a *code action provider*, which is given a document and range for which the command was invoked (see Figure 4.34). The provider must return a list of commands, where each has a command handler identifier, a list of arguments for the command handler, and optionally a title and tooltip. (Microsoft 2018c)

Cloud9

Code actions are not fully supported yet in Cloud9. The editor does not have a dedicated UI for code actions, and instead shows a tooltip indicating the keyboard shortcut through which the user can apply a single code action.

The code action handler method `getQuickFixes`, shown in Figure 4.35, is given the current caret location and AST (if available), and returns the code actions. Each code action specifies the span of text to replace in the document, and the text to replace it with. Due to the lack of UI, Cloud9 currently displays only the first code action. (Cloud9 2018d)

LSP

Eclipse Che and VS Code support code actions (called *quick assist*) through LSP. The code action handler gets a document and text range for which the command was invoked, and returns a list of commands. The API is shown in Figure 4.36. (Microsoft 2018a)

Figure 4.36: Overview of the API for code actions in LSP.

4.5.2 Feature Comparison

All editors that support code actions can replace text in the current document and display a custom label for their code actions, as shown in Table 4.37. Except for Cloud9, all editors allow arbitrary code to be executed when a code action is invoked.

	VS	Eclipse	IntelliJ	VS Code	Cloud9	Che
Abilities						
Replace text	●	●	●	●	●	●
Custom logic	●	●	●	●	○	●
Actions						
Label	●	●	●	●	●	●
Icon	●	●	○	○	○	○
Description	○	●	○	○	○	○
Tooltip	○	○	○	●	○	●
Preview	●	○	○	○	○	○

Table 4.37: Comparison of code actions editor service features across editors.
(●, supported; ○, not supported)

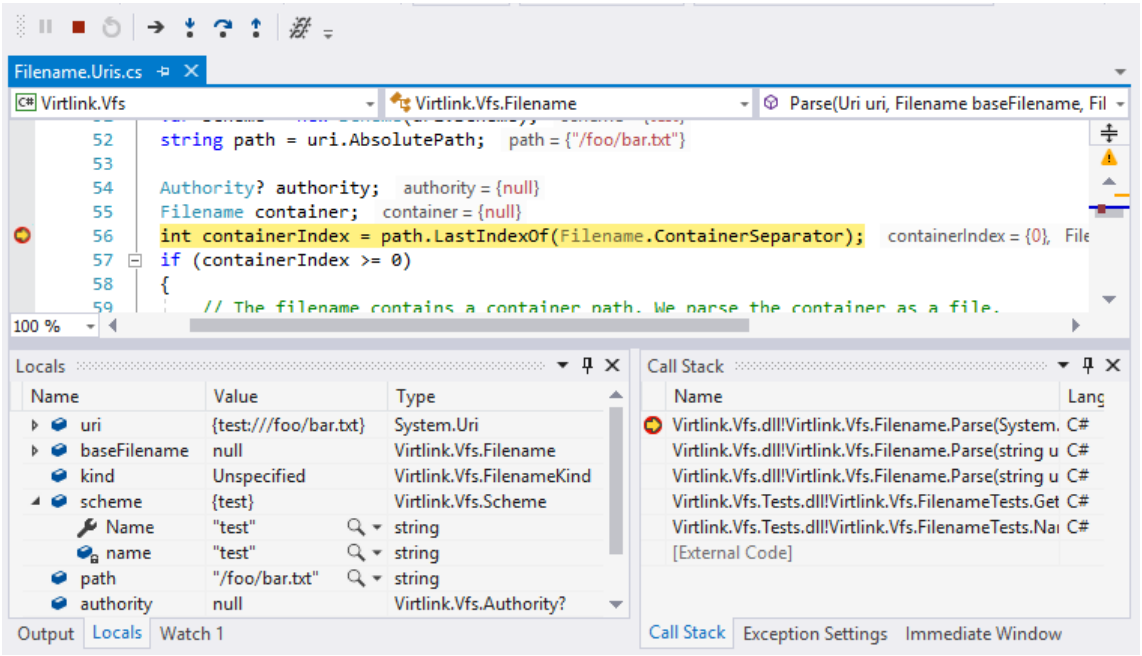


Figure 4.38: A debugging session in Visual Studio has paused execution at a breakpoint. The editor shows where execution has halted, the current call stack, and information about the local variables. Buttons at the top allow the user to perform a step, resume execution, or stop debugging.

4.6 Debugging

A debugger allows the user to pause execution of the program and inspect the program's state, as shown in Figure 4.38. Execution can be paused either immediately by user action (such as clicking the *pause* button), or when execution reaches the assigned location of a breakpoint in the code. Breakpoints are set by the user, and may carry a condition, such as to pause execution only when a variable has a particular value or when a particular exception is thrown.

Once execution is paused, the user is given an overview of the current state of the program. The editor shows the call stack, a list of functions that execution has entered but not yet exited, and the states of the various program threads. It also presents the values of local variables visible from the current execution point, and allows the user to browse the fields and properties of other objects in the program.

Some debuggers enable the user to execute arbitrary expressions and statements as if they had been in the source code at that particular point. This allows the user to change the program's state and see the effects, or to quickly inspect the result an expression would have produced.

The user can choose to resume normal execution or to only step to the next statement in the program. When the user performs a *step*, execution is briefly resumed and halted at the next statement in the same function. However, the user can choose to *step into* a function call and pause at that function's first statement, or to *step out* of the current function and pause at the calling function's next statement. Some debuggers, such as VS Code's debugger for Node programs, even allow the user to *step back*, undoing any changes and side-effects that resulted from executing the current statement. This is known as *time-travel debugging* (Barr et al. 2016).

4.6.1 Editor Support

Debuggers are an editor service traditionally founds in IDEs such as Visual Studio, Eclipse, and IntelliJ, but more recently also in non-IDE editors such as VS Code and Cloud9. Sublime, Atom, Notepad++, and Vim do not have built-in debugger support, and while the Java debugger in Eclipse Che is being worked on, support for custom debuggers is virtually nonexistent. (Eclipse Che 2016)

Visual Studio

A language can get debugging support in Visual Studio by implementing a *debug engine*. The program to be debugged is represented by a *process*, which is launched, managed, and terminated by the debug engine. A program's process is further subdivided into sets of threads that have their own memory and run independently from other sets, confusingly called *programs* in the Visual Studio terminology. Threads are the individual executions running within a program, sharing access to the program's memory, but each having their own execution stack. The execution stack is the sequence of stack frames corresponding to the function that the thread's execution has entered but not yet exited. Each stack frame has an associated collection of local variables, called *properties* in Visual Studio. A property has a value, which can have fields of its own. The relationship between these structures is shown in Figure 4.39.

A custom debugger in Visual Studio can launch any program — it does not have to be an executable file — and terminate it at the end of the session. Alternatively, the debugger can attach to and detach from an already running program. Once running, the user can control the program's execution through the standard debugger operations, if supported by the custom debugger. Those are the ability to pause or resume the program, pause or resume individual threads, and to step through the program: forward in the current function, into functions, and out of functions.

Breakpoints in Visual Studio can be used to pause program execution automatically when a condition is met. Usually a breakpoint triggers when execution reaches a particular line or statement in the code. Optionally, these breakpoints can be given an additional condition expression that must return true for the breakpoint to trigger. The editor listens to the debug engine's events, such as when a breakpoint is hit or a step is completed, and responds with the appropriate action, such as displaying the current program state.

The user can inspect the program's state while execution is paused. Each thread in the program has a stack of stack frames, each associated to a region in the code such as a function. A stack has properties, the variables in a function, whose values and child properties are displayed by the editor.

A debugger can advertise support for evaluating arbitrary expressions in the context of a stack frame. When the user inputs an expression, Visual Studio calls upon the context to parse the expression into an expression object. The expression object must implement the logic to be evaluated into a value. (Microsoft 2016b)

Eclipse

A debugger editor service implementation for Eclipse starts with a launch configuration, which can start a program and contains its configuration. The launch configuration creates a *debug target*, which is the interface from the debugger to the editor. As shown in Figure 4.40, the debug target contains the processes and threads of the program being debugged, where each thread has a list of stack frames with its variables. (Szurszewski 2003)

The various debugger interfaces in Eclipse have to implement methods for suspending, resuming, and terminating the program, adding, changing, and removing breakpoints, and

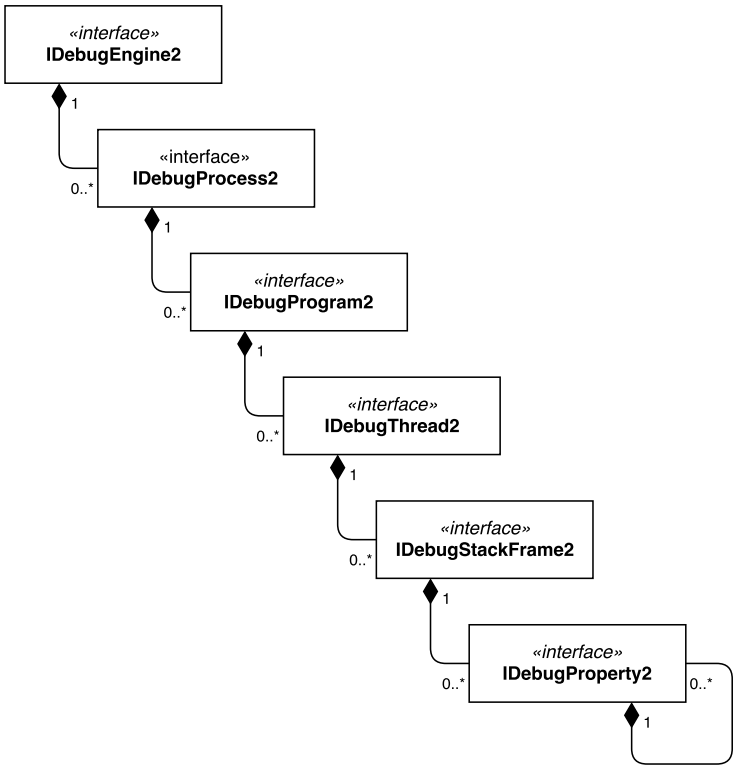


Figure 4.39: The main Visual Studio debugging API structures.

disconnecting the debugger. Additionally, the thread debug element expects you to implement step into, step over, step out of functions.

Finally, a debugger in Eclipse needs a *source locator*, to determine the exact source code location for a given stack frame. (Wright and Freeman-Benson 2004; Eclipse 2018g)

IntelliJ

A custom debugger in IntelliJ must extend the `XDebugProcess` class and implement the methods for pausing, resuming and stepping the debugger, as shown in Figure 4.41. Breakpoint handlers can be installed to record line and exception breakpoints that the user sets. The current state is represented by a tree of `XStackFrame` and `XValue` objects. (JetBrains 2015a)

VS Code

A VS Code debug *adapter* lives in a standalone executable that is launched and stopped by the editor. Communication between the editor and the debug adapter happens over `stdout` and `stdin`.

Once the debug adapter has initialized, the editor sends breakpoint configuration requests. There are three kinds of breakpoints the editor can set: normal code breakpoints, function breakpoints, and exception breakpoints. The latter two are only set when the debug adapter supports them. The debug adapter then returns for each breakpoint whether it was actually set, and where it was set to. For example, setting a breakpoint on the opening bracket of a function body may cause the debug adapter to move the breakpoint to the first statement of the function.

The debug adapter will have to raise an event whenever the program execution stops, for example due to hitting a breakpoint or when an exception occurred. The editor will then request a stack trace for the stopped thread, and for each stack trace the scopes, and for each

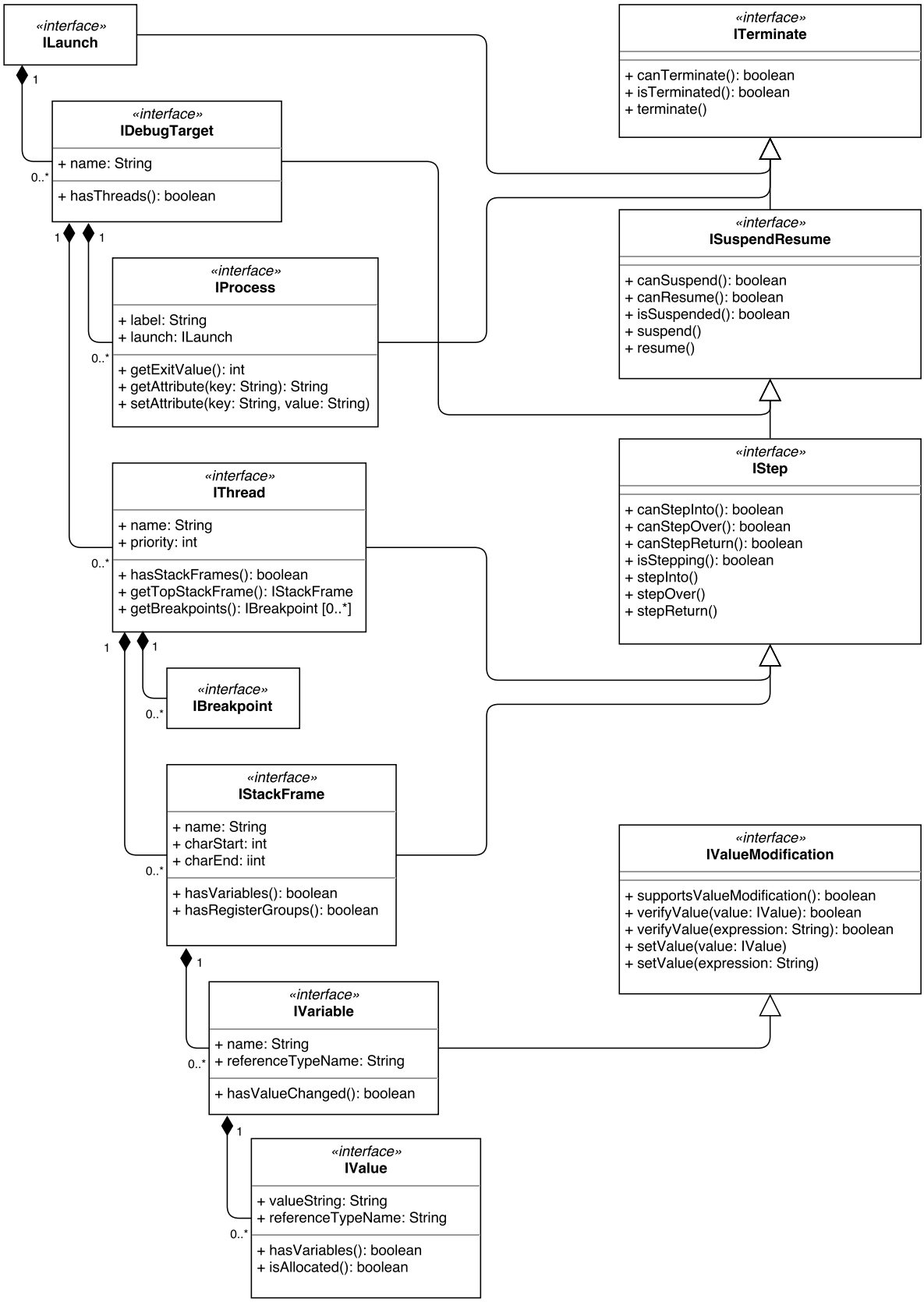


Figure 4.40: Overview of the API for debugging in Eclipse.

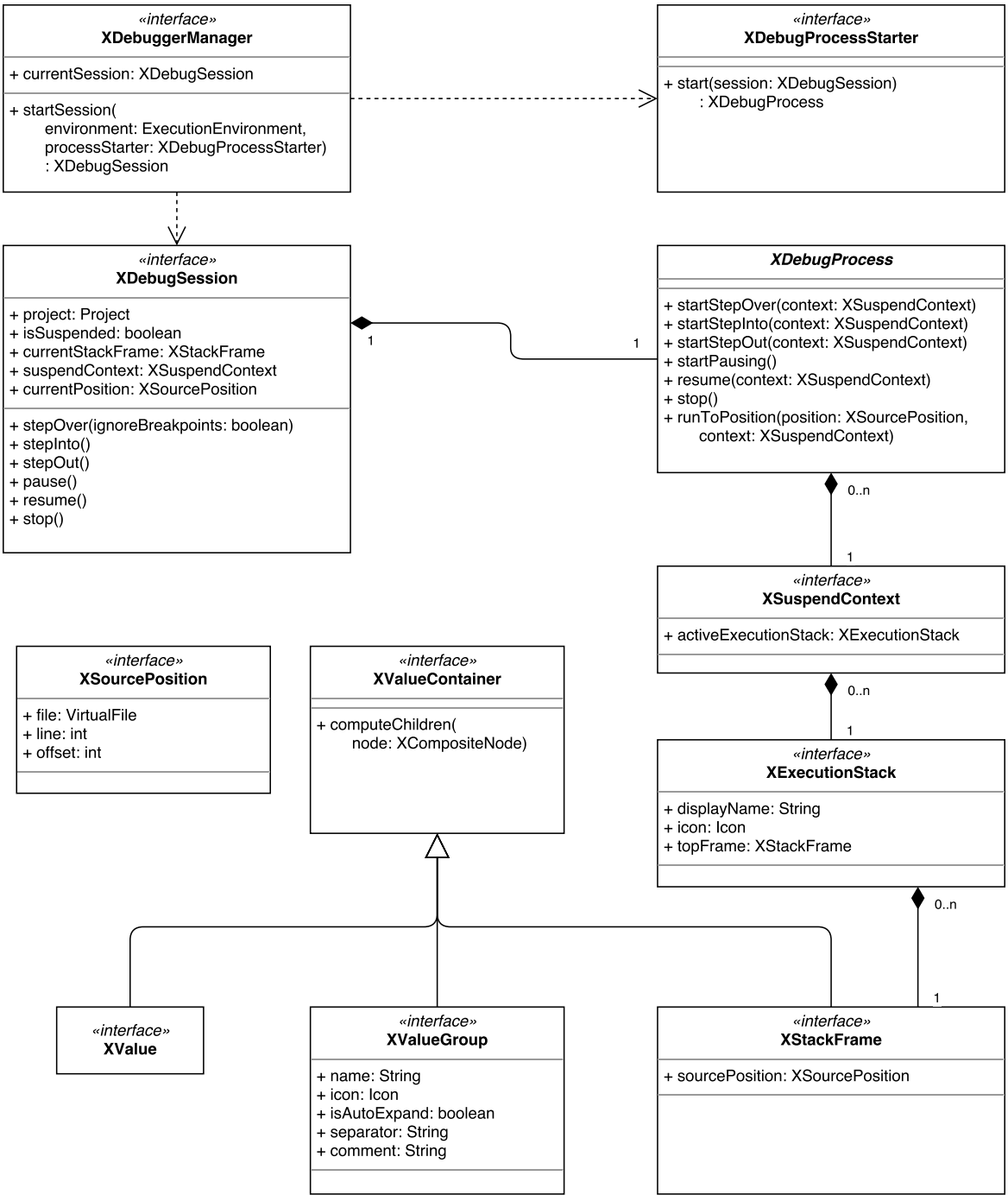


Figure 4.41: Overview of the API for debugging in IntelliJ.

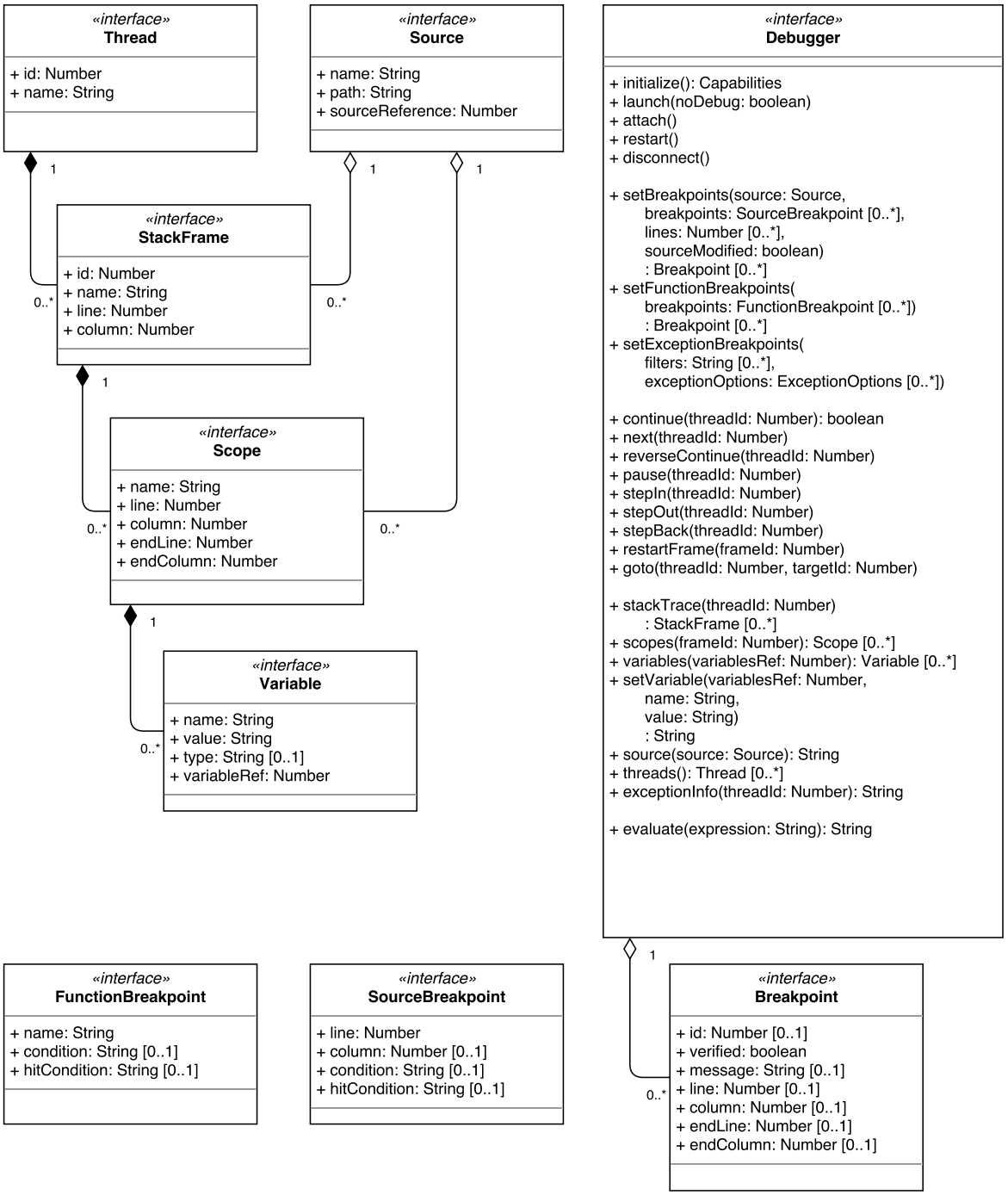


Figure 4.42: Overview of the API for debugging in VS Code.

scope the variables, and for each variable more variables if the variable is structured (for example a class or list). Finally, the debug adapter will have to handle requests to pause and continue execution, and to step into, over, and out of functions. (Microsoft 2018b) The API is shown in Figure 4.42.

Cloud9

Cloud9 supports custom runners, which are defined in a JSON file. The runner invokes the program, and optionally allows a debugger to be attached on a specific port. A custom Cloud9 debugger can support (conditional) breakpoints, live code updates, updates to vari-

ables, and a REPL implementation. (Cloud9 2018e)

A Cloud9 debugger implementation is usually stateless, and any state is stored in a custom proxy process. This makes it easier to create the debugger in such a stateless environment as a browser or cloud. The debugger must implement methods for attaching and detaching from a program, stepping into, over and out of methods, suspend en resume execution, evaluate an expression, and add remove and change breakpoints. Additionally it must have methods to return the list of source files, the list of frames on the call stack, the properties and variables in the current scope, and raise events when a breakpoint or exception is hit, or the debugger is attached or detached. Optionally, if the program uses standard I/O, a separate debug server may be needed to separate the debugger I/O from the program's I/O. (Cloud9 2018c)

The API is shown in Figure 4.43.

4.6.2 Feature Comparison

The five editors with custom debugger support have very similar features, as shown in Table 4.44. Therefore, perhaps unsurprisingly, the models of these editors are also very similar.

	VS	Eclipse	IntelliJ	VS Code	Cloud9
Operation					
Launch program	●	●	●	●	○
Custom launch logic	●	●	●	●	○
Attach and detach	●	●	○	●	●
Terminate	●	●	●	○	○
Execution Control					
Pause/resume	●	●	●	●	●
Step, step into, step out	●	●	●	●	●
Step backward	○	○	○	●	○
State					
Process	●	●	●	○	○
Threads	●	●	●	●	●
Stack frames	●	●	●	●	●
Variables	●	●	●	●	●
Miscellaneous					
Evaluate expression	●	○	○	●	●
Stack source location	○	●	●	●	●
Breakpoints	●	●	●	●	●
Conditional breakpoints	●	○	○	●	●
Exception breakpoints	●	○	○	○	○

Table 4.44: Comparison of debugger editor service features across editors with a dedicated API. (●, supported; ○, not supported)

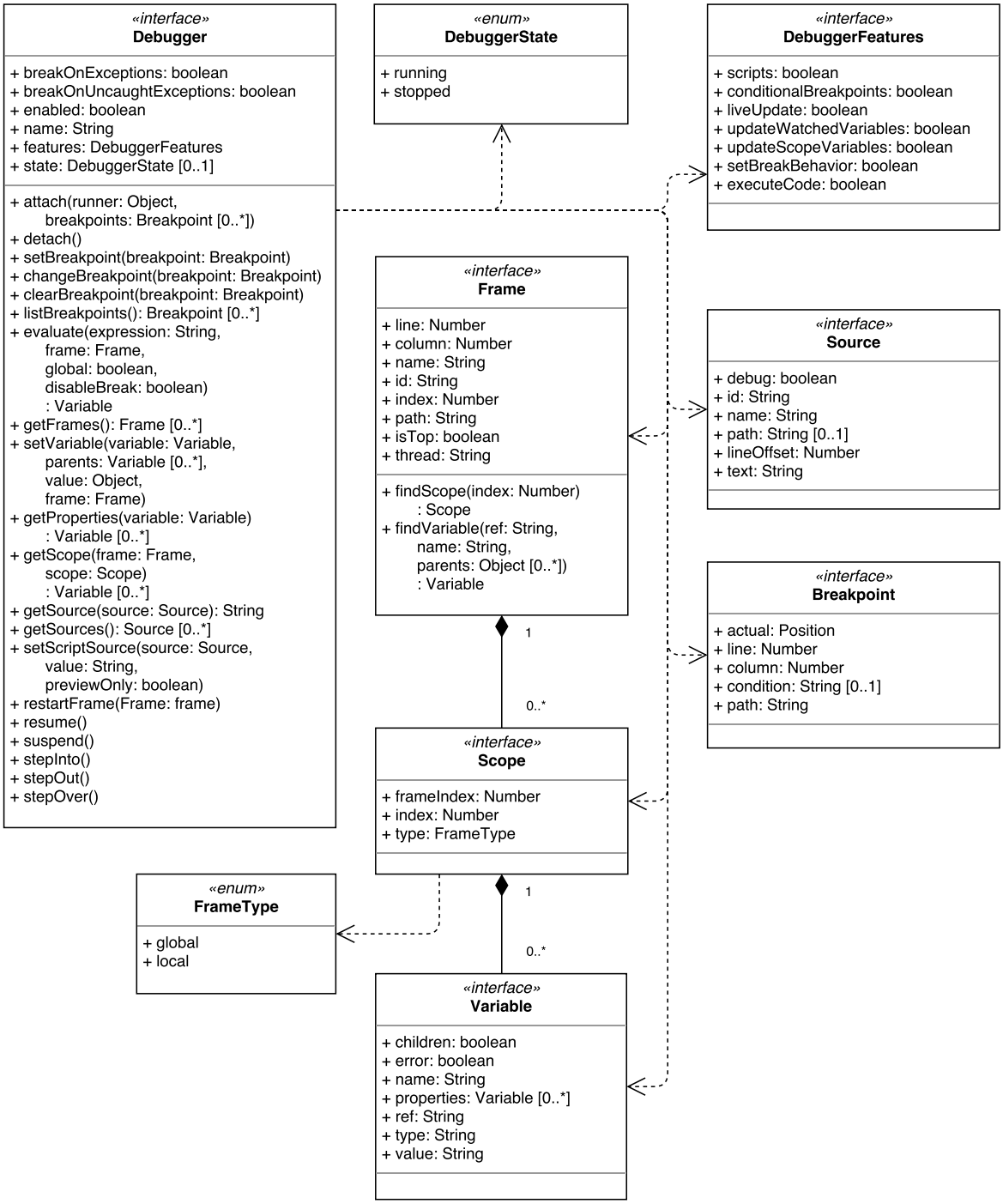


Figure 4.43: Overview of the API for debugging in Cloud9.

4.7 Miscellaneous Editor Services

While the editor services we looked at are the most important and prevalent in various code editors, there are obviously more editor services that we have not covered that are available in fewer code editors. In this section we take a quick look at some of these editor services, but we will not derive a model for them or go into detail on how to support them.

4.7.1 Compiling

Most languages can have their programs be compiled using a command-line tool. This makes it possible to integrate the language in other build systems such as Make and Gradle, and to compile the program in a *headless* environment such as a continuous integration system where no graphical user interface is available.

Many editors allow invoking such command-line compilers right from the editor, usually as part of a build script. Code editors such as VS Code (Microsoft 2018d), Eclipse Che (Eclipse Che 2017b), Cloud9 (Cloud9 2018a), and Sublime (Sublime Text 2018b) have *builders* that simply invoking this command to build the program.

Visual Studio and Eclipse have their own build script format, MSBuild and `.project` respectively. They allow the build script to invoke custom build tasks that can be implemented in code. A build task has to take some inputs and return the outputs it produced, so that these build systems can do dependency tracking and incremental builds.

4.7.2 Running

Programs written in most languages can be run from a terminal or through the operating system's graphical user interface, either directly or through an interpreter. Many editors support starting the program without leaving the editor environment, and display the program's output. Most support invoking exactly the command the user would have typed on the command-line. In some editors the user can specify additional settings specific to the language or framework, such as JVM settings or the Python version to use. Therefore, decent support across editors is already achieved when the program (or an interpreter for the program) can be run from the command-line.

4.7.3 Semantic Highlighting

Semantic highlighting is a term coined by Nolden (2009) and popularized by Brooks (2014) to denote the coloring of terms for their semantic meaning, in contrast to coloring terms by their syntactic meaning as in syntax coloring. The most prominent example of semantic highlighting is the way many code editors nowadays color local variables differently from non-local variables and fields. Some editors such as KDevelop even give different variables distinct colors, as shown in Section 4.7.3. There is a proposal to add semantic coloring to the Language Server Protocol (LSP) in the near future (Efftinge 2016).

4.7.4 Safe Delete Refactoring

Safe delete is a refactoring supported by IntelliJ in addition to rename refactoring. When the user attempts to delete a symbol, such as a variable or a class definition, the code editor will determine whether all usages of the symbol are safe to delete. A usage is safe to delete if deleting it has no effect on the semantics of the code, such as when a class is only imported but never used, or when a local variable is only ever written to and never read from. If unsafe usages are found, the user is given an opportunity to review these usages and make the necessary changes. (JetBrains 2018m)

```
double a, b, c;
double determinant;
double root1r, root2r, root1i, root2i;

determinant = b*b-4*a*c;

if (determinant > 0) {
    root1r = (-b+sqrt(determinant))/(2*a);
    root2r = (-b-sqrt(determinant))/(2*a);
} else if (determinant == 0) {
    root1r = root2r = -b/(2*a);
} else {
    root1r = root2r = -b/(2*a);
    root1i = +sqrt(-determinant)/(2*a);
    root2i = -sqrt(-determinant)/(2*a);
}
```

Figure 4.45: Semantic coloring of variables in the KDevelop editor gives every variable a different color.

4.7.5 Code Lens

Code lens is an editor service found in Visual Studio and VS Code where textual metadata about the code is printed in between the lines of source code. This metadata can include among other things the author, number of references to a function, the test suite coverage, and the time of last change. (Anderson 2017)

Code lens is part of the Language Server Protocol (LSP), such that while more and more code editors adopt support for LSP, the code lens editor service will become more ubiquitous. (Microsoft 2018a)

4.7.6 List Symbols

With detailed symbol metadata available for the reference resolution and structure outline editor services, there is an opportunity to allow users to list the symbols in the current document or workspace and search for symbols of a specific type. The ability to list symbols is present in various editors, such as the LSP-enabled editors VS Code and Eclipse Che. (Microsoft 2018a)

4.8 Conclusion

Most editor APIs do not significantly restrict the implementations of their editor services, with IntelliJ being the notable exception. IntelliJ uses its PSI internal structure for most of its editor services. Implementations can use the PSI structure to their advantage, as it provides built-in facilities to easily support, among others, the structure outline, reference resolution, and code completion. However, this restricts implementations of the editor services when they cannot take advantage of the PSI structure.

While editors distinguish themselves through their features, there is a core set of editor services that is widely supported across editors. The APIs of these editor services show they are also similar in their implementation across editors, but not similar enough that a single implementation with slight modifications would suffice. These commonalities can, however, drive the design of a generic interface for editor services, as shown in Chapter 5.

Chapter 5

Adaptable Editor Services Interface

AESI, the Adaptable Editor Services Interface, is our contribution for an editor services model that provides a solution for the IDE portability problem of having to write and maintain a binding for every combination of a language and editor. Instead, a language implementing the AESI model can be used from an AESI-enabled editor with little effort.

Editor services are provided by the editor and help the user reading and writing code, but the support, features, and requirements of an editor service implementation can differ across editors. A programming language that wishes to support these editors would have to adapt the editor services of every editor to their implementation. If instead the editor services were adapted to a common interface such as AESI, languages need only support the common interface to support multiple editors.

We modeled the interfaces in AESI such that they fit within the requirements and restrictions imposed by the most popular editors (see Chapter 4), to ensure that they can be adapted to a wide range of editors. For example, while many editors can show icons in their interfaces, most only show built-in icons, which means the AESI model cannot assume editors can load arbitrary icons from disk.

We adopted the most relevant editor services into the AESI model, derived from the model of important language workbench features identified by Erdweg et al. (2013); the same editor services we explored in Chapter 4. These are: syntax coloring, code folding, code completion, structure outline, symbol navigation, hover documentation, signature help, automatic formatting, rename refactoring, code actions, diagnostic messages, debugging, and testing.

AESI was motivated by our implementation of the Spoofax language workbench in the IntelliJ editor (see Chapter 2), where the continuous evolution of Spoofax also influenced our Spoofax adapter for IntelliJ due to how intertwined the adapter is with Spoofax. AESI provides a way to separate the concerns of the workbench from its editor service adapters.

In this chapter we show and motivate our design for AESI, a platform- and language-agnostic editor services interface. Section 5.1 lays out the requirements of the AESI model. In Section 5.2 we show the general motivation behind the model's design, and any editor service agnostic concerns. Sections 5.3 to 5.8 detail the models for the six most interesting editor services, namely syntax coloring, code completion, structure outline, reference resolution, code actions, and debugging; whereas the rest is covered in Appendix B. In Section 5.9 we evaluate the AESI model through our implementation of four editor services in three editors and two languages.

5.1 Requirements

AESI is a way for editors (through an adapter) and language implementations to talk to one another. The primary goal is to enable portable editor services, where an implementation of

the editor service interface can work without modifications on a different editor that has an adapter for the interface.

As part of this portability goal, the interface should impose the minimum requirements on implementations. This keeps the interface free from the specifics of any particular editor, and would make it easier for a language to provide an implementation. For example, the syntax coloring editor service in most editors needs to return only a list of text ranges and their associated styles, while IntelliJ expects a complete abstract syntax tree of the file. Rather than imposing the requirement of a syntax tree on every implementation, we choose to require only the minimum that most editors need (a list of text ranges and associated styles) and put the burden of converting this representation to IntelliJ's abstract syntax tree on the editor adapter.

A language implementation may choose to provide only a minimal implementation, or not support an editor service interface at all. This allows an implementation to gradually improve its editor services support, without having to supply a full implementation upfront.

Similarly we do not want to force an implementation to use a particular communication protocol. When thinking of portable editor services, one concern that comes up is the ability to write the editor service implementation in a language or framework compatible with the language itself, which is often different from the language or framework used by the editor. A common solution for this, as used in the Monto framework (Sloane et al. 2014) and the Language Server Protocol (Microsoft 2018a) is to always require a particular communication protocol such as ZeroMQ or JSON RPC, as this decouples the editor service caller from the editor service implementation. What communication protocol is appropriate, or whether to use one at all, can depend on requirements of responsiveness, complexity, and distance between the editor and the language's editor services. We discuss this further in Section 5.2.3.

Additionally, we want to support a big feature surface that includes those features that are provided by more than one editor. For example, if a code completion proposal can have an icon in some editors, the language implementation should be able to (but is not required to) provide such an icon. Those editors that support it can show the icon, and those that do not support it can ignore it. This is known as the *least common multiple* approach, which is one of the two portability approaches identified by Feldman and Gentleman (1990), where we model all the desired features and functionality and let each editor handle a subset. The other approach is the *greatest common divisor* approach, where the model supports the minimal functionality supported across all editors and each editor can augment the functionality with extra features. We chose the *least common multiple* approach as we argue it is easier to ignore some functionality or features than to add them retroactively.

To summarize, the design of the AESI model will have to adhere to the following requirements:

1. editor services portable across editors;
2. expose maximum features supported by more than one editor;
3. impose minimal requirements on language implementations;
4. design for use with a communication protocol without enforcing one.

5.2 Design

This section covers the concepts and design decisions that are independent of any particular editor service. They are required for a complete understanding of AESI.

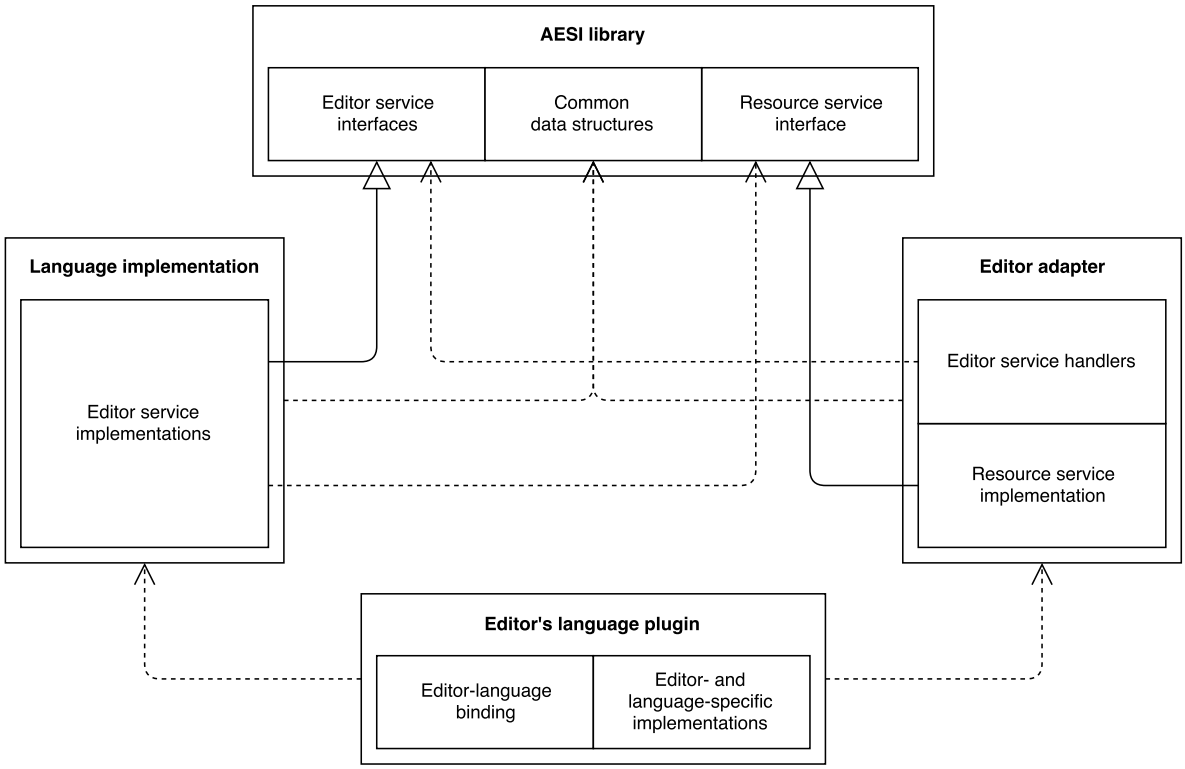


Figure 5.1: Diagram of the dependencies between a language and editor with AESI. The adapter calls the AESI editor service interfaces which are part of the language implementation, and the language implementation calls the AESI resource service which is implemented by the editor adapter. The editor's language plugin binds the language implementation and editor adapter together, and integrates them into the editor.

5.2.1 Architecture

The architecture of AESI is depicted in Figure 5.1, showing the relations between languages and editors. A *language implementation* is the concrete implementation of the AESI editor services for a particular language. An *editor adapter* calls these implementations when handling the editor services for a particular editor. The adapter also exposes a *resource service*, which is used by the language implementation to access the content of the documents being edited. The language implementation and editor adapter are bound together in a plugin for that particular language-editor combination.

The separation between language implementations and editor adapters means that for a particular editor only one editor adapter needs to be written and maintained to support any AESI-enabled language. Similarly, for a particular language only one implementation has to be written and maintained to support any AESI-enabled editor. Both will use an AESI library with the common interfaces and data structures. The final piece of the puzzle is the plugin that ties the language to the editor, which is a small project that just binds the AESI interfaces to their implementation. We will now discuss each of these main components.

AESI Library We express the model of AESI in terms of a set of interfaces that describe the interaction between the various language implementations and editor adapters. In practice, these interfaces are implemented in a library, along with some data structures and helper functions that are commonly used by both language implementations and editor adapters. Both language implementations and editor adapters depend on an AESI library.

Language Implementation A language implementation provides concrete AESI editor services. The editor service interfaces impose minimal requirements upon their implementations, yet expose a full set of features supported by various editors. A language implementation can choose to support only some of the features, giving it full flexibility in implementing the service. When a language cannot provide a particular editor service, its implementation simply returns no results when it is used.

Editor Adapter For each editor service, it is the job of the editor adapter to call the AESI interfaces when an editor service is invoked, and to translate the return values to a format the editor understands. Additionally, the editor adapter must expose the latest versions of the editor's opened documents to the language implementation through its resource service.

Editor's Language Plugin The implementations provided by a particular language and editor are tied together in a plugin, which is loaded by the editor. While ideally any AESI language can be instantly loaded in the editor of choice, in practice this is difficult.

Firstly, most editors can load plugins only at startup, and often a plugin can contribute only a single language. While there are ways to circumvent these restrictions, such as using undocumented application programming interfaces (APIs) in IntelliJ or representing all AESI languages as a single language in Eclipse, it is not clear that this is possible in all editors.

Secondly, it would significantly increase the complexity of the editor adapters. Every adapter would need to provide its own language loading logic next to the logic already present in the editor, and track the languages of the files in the editor. The editor adapter would also have to handle concurrent use of multiple languages and their editor services.

Instead, we chose to have a plugin for each combination of editor and language. This plugin is a very thin wrapper around the language and editor adapter, and can easily be generated. For example, dependency injection can be used to associate the AESI interfaces with their language-specific implementations, and expose these to the editor adapter.

There are some advantages to having a dedicated plugin. The combination of a language implementation and editor adapter that are known to work together are kept together, such that a language plugin keeps working even if a newer version of the editor adapter has backwards incompatible changes. An editor may even have different versions of the editor adapter running side-by-side, without them influencing one another. Likewise, a plugin dedicates an editor adapter to a single language, removing the need to support concurrent operations of editor services of different languages.

Finally, sometimes it may be preferable to add behavior specific to the combination of the language and editor to the plugin, in order to provide better integration with a particular editor than the editor-agnostic AESI model can provide. For example, while AESI aims to support a full set of features, there will always be editor-specific features that are not supported by AESI, either because they are too editor-specific, or because they were later added to an editor. Such language-editor-specific code will have to be maintained, which is a trade-off, but this burden should be minimal, as most of the implementation is already maintained as part of the general AESI editor adapter.

5.2.2 Resource Service

Many editor services apply to the document the user is currently looking at, and therefore need to know the exact text of the document, such as code completion suggesting keywords and identifiers based on what the user just typed. Furthermore, editor services may need the source code from other documents in the same project, for example to provide reference resolution to definitions in other files. Because of this, the editor services need access to a file system. An obvious choice is to simply use the local file system, but not only will this prohibit remote file editing such as FTP or cloud-based editors, it would actually be incorrect:

a document that has just been modified or created in the editor, but not yet saved, does not yet have its latest content stored in the local file system.

The true content of a document is determined by its *source of truth*. While a document is being edited, the editor is the sole source of truth. In other cases the source of truth can be the local file system, a remote file storage, or a database.

For AESI we need a virtual file system, which is a file system that editor services can read from and browse, but which abstracts over the actual sources of truth. For example, one virtual document might represent an unsaved document in the editor, whereas another document is represented by its content on disk, a third document is found only inside an archive, and a fourth is on some cloud storage.

The *resource service* is our interface for this virtual file system. It identifies every document and folder by its Uniform Resource Identifier (URI), which is opaque to the language implementation. The resource service can tell what kind of resource a particular URI represents, whether it currently exists, and what children or content it currently has. A class diagram of the `ResourceService` API is shown in Figure 5.3.

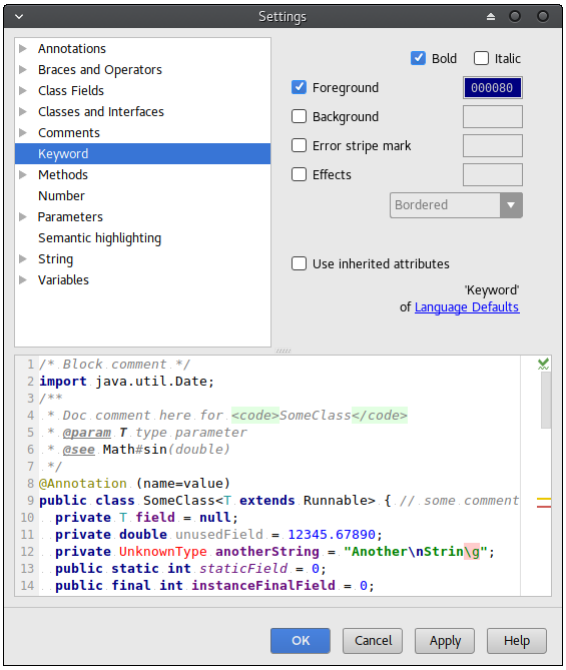


Figure 5.2: Color scheme editor in IntelliJ. The bottom document is a read-only document without a filename.

An implementation of the resource service is provided by the editor adapter, which allows it access to details specific to the editor being adapted, such as whether a document is currently opened in the editor and what its current unsaved content is. It can also represent documents that have no name yet, such as newly created but unsaved documents.

The resource service can provide the content of a given document, which is an immutable snapshot of the document's content at the time of the call. The snapshot is identified by its last modification stamp, a number that is guaranteed to be different for different versions of the same document in the same session (see Section 5.2.4). The string content can be read through either a call to `getText` to get the full text, or more efficiently through a `Reader` (or equivalent).

The content object has functions to get the full text of the document, the number of lines, and functions to convert an absolute document character offset into a line-column-based location and vice versa. We put this functionality in the content object itself, as many editors already have an efficient internal representation of the lines in a document and the means to convert between absolute and line-based location within the document. The content object, whose implementation is provided by the editor adapter, can just reuse that functionality if available.

Most notably, this API has no functionality to change the content of a document. We figured this would not be needed as all editor services that change a document, such as the automatic formatting service, already return the list of changes to apply to the document. It would also complicate the implementation of the resource service, as it has to deal with content that changes outside the control of the editor. Finally, some documents are read-only, such as the preview document of the color scheme editor in IntelliJ, as shown in Figure 5.2.

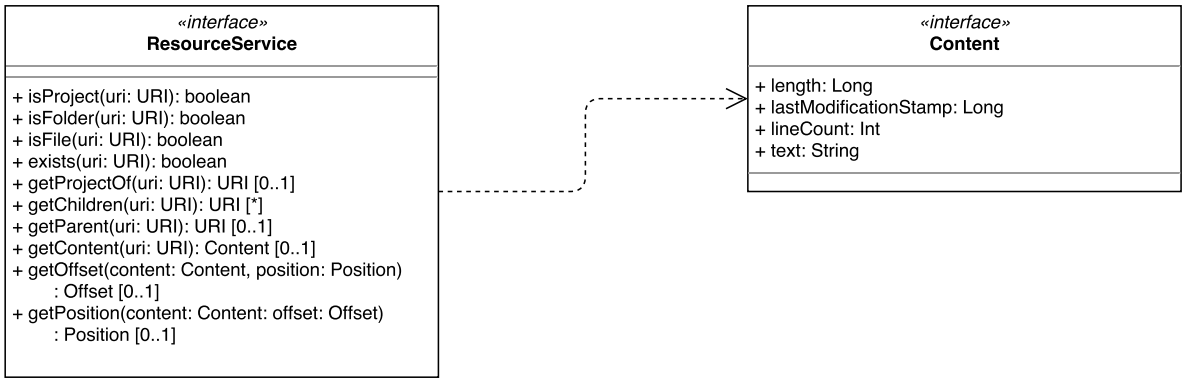


Figure 5.3: AESI resource service API.

Consistency

We require that the documents returned by the resource service are internally consistent: a document's content is represented as it has actually been at some point in time. A document can become internally inconsistent when, for example, a service sees only the modifications applied after what has already been read. This results in a document state which has never actually occurred.

Contrast this to external consistency, which is when documents have to be consistent among themselves. It is easy for multiple documents to get temporarily inconsistent: one document has been changed but not yet saved in one editor, while the editor has not yet picked up on changes made by another program to a related document on disk.

As an editor service cannot predict which documents it might need, and cannot read the content of all documents in a single instant, it would require a consistent *view* of all documents it might ever need. With multiple data sources, such as dependencies that are in a remote repository, or documents that are stored in the cloud, it becomes infeasible or even impossible to enforce a consistent view of all these data sources.

Even if an editor service could work on a consistent view of all the documents, a single change to any document, even if irrelevant to the service, would make any results of the editor service no longer applicable to the current, changed, view of the documents.

Therefore, we do not require external consistency for the documents in our model. Eventually, when all documents cease to be updated, they converge to a consistent state. In the mean time, an editor service may operate on an externally inconsistent view of the documents. When necessary, the editor service is called again for the now changed documents.

5.2.3 Communication

The communication between the editor adapter and the language implementation is intentionally unspecified. When the language implementation and editor run in the same process and use the same platform, such as Java, then it makes sense to load the implementation with the plugin and call the interface methods directly. This provides good performance with minimal overhead.

If, however, the language implementation and editor use different platforms, then a communication protocol is needed to bridge this gap. The choice here depends on the requirements of both. A general solution would be to communicate over a socket connection, which would work regardless of the distance between the editor and language implementation. However, it introduces an unacceptable overhead when both the implementation and the editor run on the same system or even in the same process. In that case, communica-

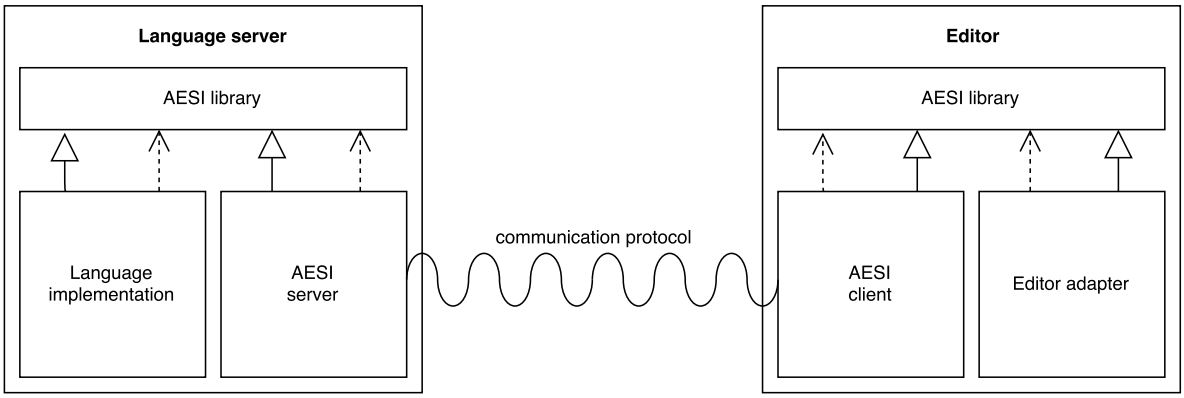


Figure 5.4: Custom communication between editor and language. An AESI server and client take the places of the editor adapter and language implementation respectively, and relay the requests and responses to one another.

tion over the standard input/output streams makes more sense, or use a solution such as ZeroMQ[1] (Sústrik 2012) or NanoMsg[2] that provide asynchronous messaging for a variety of situations. When an editor has built-in support for a communication protocol, such as Language Server Protocol (LSP) in VS Code, then it makes sense to use that instead.

As there is no single communication solution superior in all scenarios, we argue it makes sense to leave the communication as a separate concern. Figure 5.4 shows how a communication client takes the place of a language implementation at the editor, but instead relay the requests to a server. Likewise, a language server would implement the AESI interfaces as if it had an editor adapter, but instead respond to the client's requests. This way both the language implementation and the editor adapter are independent from how they communicate.

5.2.4 Model Design

The AESI model is a description of a number of interfaces. Considerations that impacted the design of these interfaces are outlined in the sections below: document offsets, cancellable operation, extensibility, session management, scope names, text styling, text edits, and commands.

Document Offsets

Most editors report locations in a document in terms of the absolute character-based offset from the start of the document, whereas some editors report locations in terms of a line number and column offset from the start of that line. The latter is most useful for the editors themselves, when they internally maintain each document as a sequence of lines. However, absolute offsets are easier to work with for editor services. For example, the length of a region between two offsets can be readily determined without reading the document, which is not the case for a region between two line-column-based locations when they span more than one line.

The conversion from absolute offsets to line-column locations and vice versa is trivial if the document's content is known. Given that most editors report absolute offsets, we chose to express all document locations in AESI in terms of absolute offsets. However, for convenience

[1] `http://zeromq.org/`
[2] `http://nanomsg.org/`

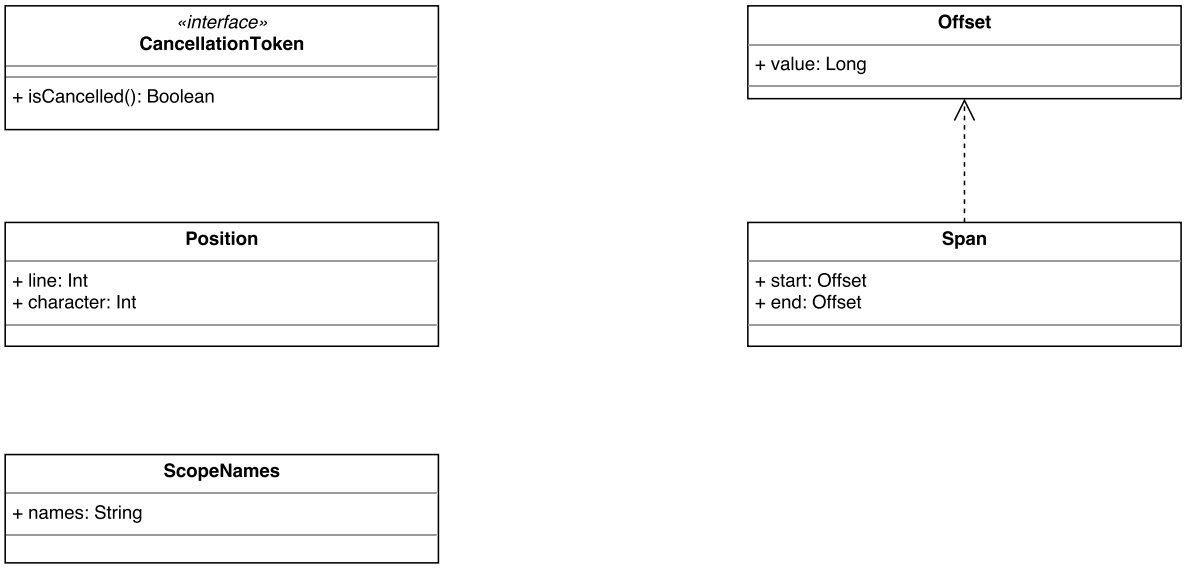


Figure 5.5: Class diagram of AESI supporting classes and interfaces: offsets, locations, spans, cancellation tokens, and scope names.

AESI also has a data structure to represent line-column locations. Both classes are shown in Figure 5.5. The `ResourceService` object, shown previously in Figure 5.3, has methods for converting between line-column locations and absolute offsets.

Cancellable Operation

An editor service implementation may take some time to compute the results of a request, and is therefore often executed asynchronously to prevent blocking the editor. However, this also means that events that occur while the editor service is computing its results can render any results invalid. For example, when the user changes the text for which the code completion editor service is computing the list of proposals, the returned proposals would no longer be valid. If the result is going to be ignored anyway, the request might as well be cancelled.

Every editor service accepts an optional *cancellation token*: an object which the service can query to see whether the operation has been cancelled, as shown in Figure 5.5. It is entirely valid for the editor service to ignore the cancellation token, or for the editor adapter not to provide one, in which case the request will just complete whether the results are used or not. However, if the editor service does respond to the cancellation token, it can stop its operation to free computing resources for other tasks.

The actual implementation of the cancellation token object is provided by the editor adapter. It may adapt an editor's existing cancellation mechanism, or implement their own thread-safe cancellation token.

Extensibility

While AESI allows a language implementation to support many editors out-of-the-box, sometimes this is not sufficient. A specific editor can have or gain specific features that are not supported by AESI, or a language needs better editor service integration than what a generic editor adapter can provide. To support these cases, we left room for extensions.

The main way to extend AESI is to extend its interfaces. An extended interface recognized and used by both the editor adapter and the language implementation can add new abilities

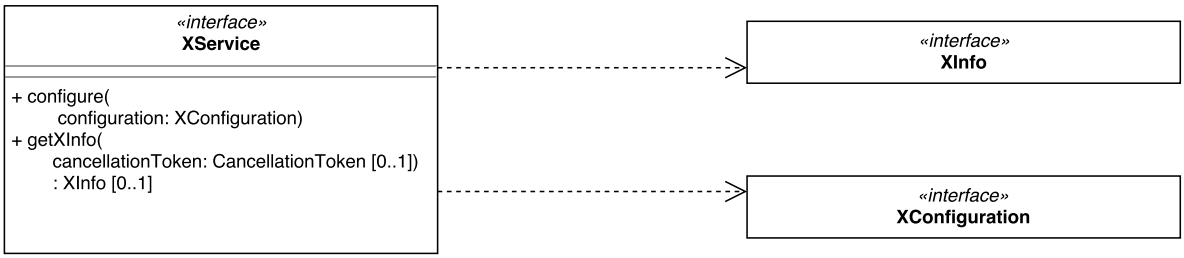


Figure 5.6: Template for most AESI editor services, here shown for some service *X*.

to the editor service, add configuration parameters, or return additional data.

To this end, the interfaces of each editor service follow the template shown in Figure 5.6. The editor service, represented by the `XService` interface for some service *X*, can be configured through the `configure` method that accepts an `XConfiguration` with the configuration parameters. While most AESI editor services do not have any configuration parameters, by extending this interface the configuration can be extended. For example, the configuration of the structure outline service is empty by default, but an extension could add a maximum tree depth configuration parameter here.

The main methods of each AESI editor service return either nothing, in the case of failure, or a single object, represented by `XInfo` in the figure. Even editor services that return multiple results return them as part of this single object, so that an extension wishing to add return data can do so by extending the returned interface. As an example, the reference resolution editor service could be extended to return an attribute indicating the primary definition from all definitions it normally returns.

Session Management

An editor service can choose to be incremental, skipping the computation and immediately returning the previous result when there are no changes to any of the documents on which it depends. To allow for this, we need an easy way to distinguish different versions of the same document. There are several solutions to do this.

One solution is to save the document content on which a result was based, and compare it with the current document's content. While this is very exact, the space and complexity are in the order of the sizes of the documents. Alternatively, by calculating and storing a hash for the document we can reduce the space requirements while still keeping the exactness of the comparison. Finally, with minimal space and complexity requirements we can track and compare a document's last modification time or version, but this may indicate that a document has changed even if the content is still the same.

For AESI we chose to track a document content's last modification *stamp*, similar in function to a last modification time, which is an integer value that is equal when the content is guaranteed to be equal. This, however, does not imply that the content is different when the stamp is different, which may result in some superfluous computation for incremental editor service. For example, when the user undoes its changes to a document, even though the document has the same content as a previous version of the document it still gets a new modification stamp. Of course, an editor service is still free to compare documents in another way, such as through calculating their hash.

The last modification stamp is unique to the current session, where a session starts when the user opens the project in an instance of the editor, and ends when the project is closed. This is to make the implementation easier, as a change to a document's content could increment the stamp without having to consider stamps used in previous sessions or sessions

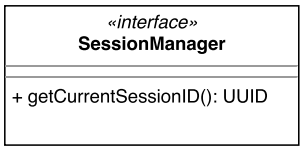


Figure 5.7: Model for the session manager.

that run next to the current session, such as when the same project is opened in two different editor instances. However, this means we need a way to identify the current session and distinguish it from other sessions. For this we provide the *session manager*, as shown in Figure 5.7, which simply returns a unique identifier of the current session. We chose for the identifier to be a Universally Unique Identifier (UUID) (The Open Group 1997), as they can be easily generated independently, and are extremely likely to be unique.

Scope Names

Editor services often want to control the styling of text, such as in syntax coloring, or the icons shown in the interface, such as next to code completion proposals or elements in the structure outline. However, we argue that an editor's presentation, such as the text styles and interface icons shown, is an editor concern and not a language concern.

The styling of source code can depend on the editor's theme (such as a light or dark theme), the editor's installed color schemes, and the custom color configuration set by the user. On top of that, not all editors support the same styling operations: Vim, for example, cannot display text in a bold typeface.

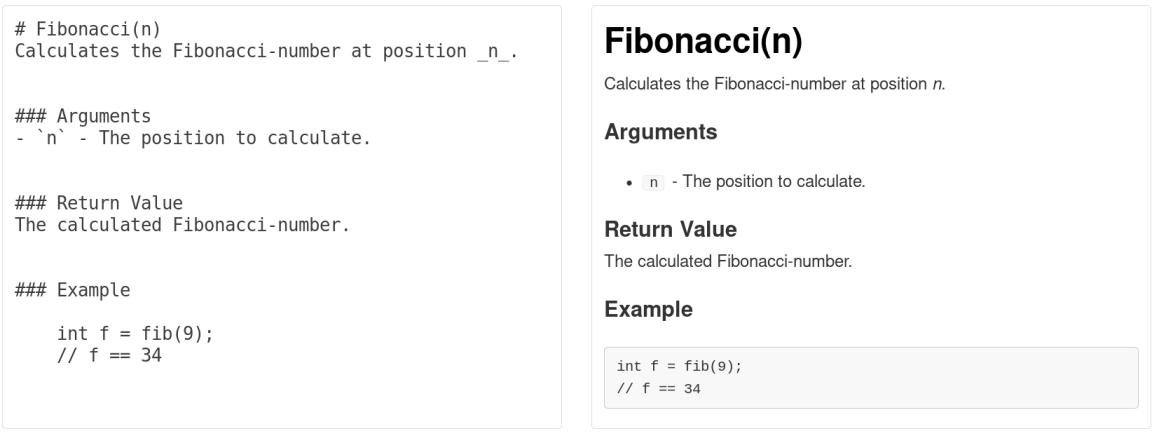
Similarly, the choice of icons must not conflict with user interface concerns such as the background color — using a dark icon on a dark background would render it nearly invisible to the user — and should ideally match the general theme of the editor's icons. Finally, some editors have a predefined list of icons from which editor services must choose, and do not support arbitrary icons.

Language implementations need a way to influence the style of text and the icons shown, but cannot specify these directly. We could try to define a list of possible syntax token kinds, but this list can never be exhaustive or support as-of-yet unknown constructs in future languages. Instead, we propose to appropriate the mechanism of using *scope names*, as used in TextMate grammars. A scope name, previously shown in Section 4.1.1, is a sequence of increasingly more specific identifiers, each separated by a dot. It is the job of the editor adapter to interpret the scope name and pick the appropriate text style and icon. For example, a token with scope name `keyword.operator.arithmetic.java` indicates that it is an arithmetic operator keyword in the Java language, but lacking coloring for this specific kind of token, the editor adapter can fall back to a more generic coloring, such as the coloring used for all operator keywords.

Most scope names are used to assign syntactic categories to tokens in the source code, but scope names can also be used to convey semantic information (such as whether an identifier represents a class or a function, and whether its use is deprecated). While language implementations are free to choose any scope name, we recommend picking scope names from the comprehensive list in the Sublime documentation [3]. Editor adapters should use the same list to find appropriate styles and icons for the objects returned by the editor services.

As scope names convey both syntactic and semantic information, it is possible for more than one scope to apply to a given object. For example, a code completion proposal can be an identifier, that represents a class, and which is deprecated. For this reason, we allow multiple

[3] `https://www.sublimetext.com/docs/3/scope_naming.html`



```
# Fibonacci(n)
Calculates the Fibonacci-number at position _n_.


### Arguments
- `n` - The position to calculate.


### Return Value
The calculated Fibonacci-number.


### Example

    int f = fib(9);
    // f == 34
```

Fibonacci(n)

Calculates the Fibonacci-number at position *n*.

Arguments

- `n` - The position to calculate.

Return Value

The calculated Fibonacci-number.

Example

```
int f = fib(9);
// f == 34
```

Figure 5.8: Markdown example, showing the raw Markdown text on the left, and the resulting styled text on the right.

scope names to be specified for the objects in our model, represented as a comma-separated string in the `ScopeNames` object shown in Figure 5.5.

Text Styling

Some AESI editor services, such as the hover documentation service, support styled text, where bold and italic types, and monospaced fonts can be used to show emphasis and code samples, and improve readability. The way text is styled differs from editor to editor. VS Code and Eclipse Che use the Markdown markup language, whereas Cloud9 uses HyperText Markup Language (HTML). Other editors use their built-in user interface (UI) components to display styled text.

Markdown is a styling language that uses special characters to indicate the style of the text. For example, surrounding a word with asterisks makes it **bold**, whereas indenting a line with four spaces makes it a `monospaced` code block. A big advantage of Markdown, compared to other styling languages such as HTML, is that even as raw text without styling it is still quite readable, as can be seen in Figure 5.8. A disadvantage of Markdown is that there is not one standard specification for the syntax; many implementation exist and they differ in features and the handling of corner cases. One attempt to standardize Markdown is the Commonmark project[4], which aims to provide a complete and unambiguous specification of the Markdown language.

Wherever we support styled text in AESI, we chose Commonmark as the styling language, as it is supported by some editors already. Those editors that do not support it can either display the raw Commonmark text as-is, strip it down to bare text, or convert the Commonmark syntax to their own styling representation, such as HTML.

Text Edits

Some services, such as the rename refactoring and automatic formatting services, have to edit the documents in the project. While we could choose to have the service send the full documents resulting from the edits to the editor, this would impose a large overhead, especially when the edits concern many or very big documents. Instead, we describe the changes as a collection of *text edits*, inspired by the text edits in LSP.

Each text edit selects a span of characters in current version of a document, and specifies the text to replace it with. When the replacement text is empty, the selected characters are

[4] `http://commonmark.org/`

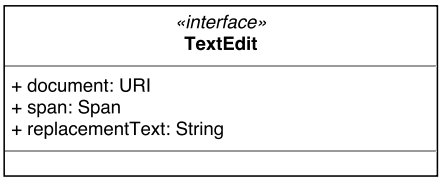


Figure 5.9: Model for text edits.

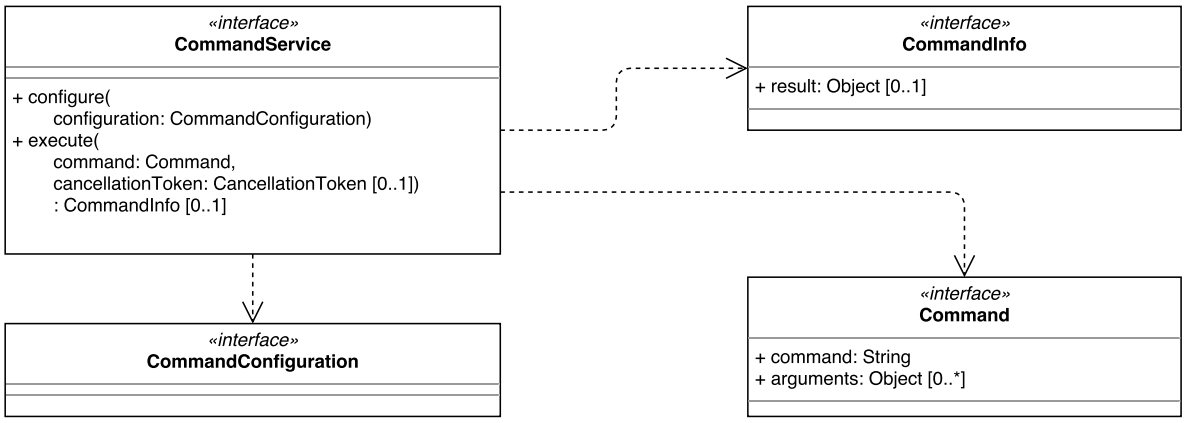


Figure 5.10: Model for the command service.

just removed. Similarly, when the selection has no length, the characters are added. All the offsets in a text edit are in terms of the original document, to ensure that a collection of text edits can be applied in any order to the current version of a document. The `TextEdit` class is shown in Figure 5.9.

Commands

Through the *code actions* service, the user can execute a command proposed by the language implementation. These commands can have arbitrary implementations, provided by the language implementation, and not known in advance.

The AESI `CommandService` can execute these commands, whose model is shown in Figure 5.10. It accepts a command string with arguments, and executes it when the command is known and valid. When executing the command result in changes to a document, the command service returns a collection of `TextEdit` objects that will be applied by the editor.

In the following sections we will discuss the AESI model for each of the editor services, and their design and motivation.

5.3 Syntax Coloring

Only four of the editors we have looked at provide programmatic support for syntax coloring, namely Visual Studio, Notepad++, Eclipse and IntelliJ. All the other editors except Vim could be supported with a TextMate grammar, but this would have to be generated from a syntax definition or created manually.

There are two ways in which the four editors implement syntax coloring. IntelliJ builds an abstract syntax tree (AST) for the file, and determines the coloring of each token by the type of its AST element. The other three editors associate styling information with ranges of source code, usually single tokens. This is also the approach we take in our model, where the coloring service returns the syntax information as a list of tokens. This is a simple mechanism,

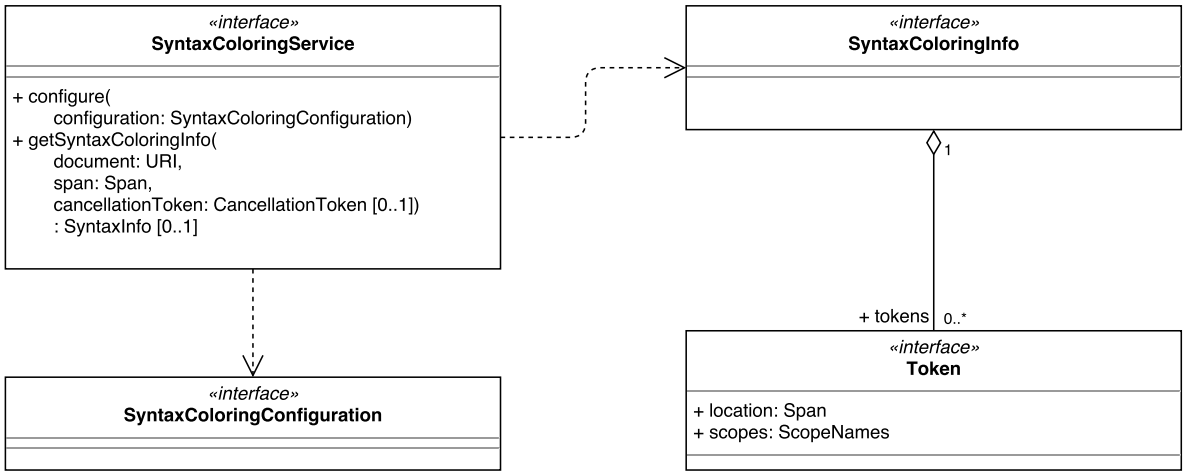


Figure 5.11: Overview of the AESI API for syntax coloring.

which we believe can be adapted to work with IntelliJ as well. One way would be to build a shallow AST which represents the sequence of styled ranges.

Since these editors all support color schemes, we support that in our model as well. If we were to specify the exact color and style of each token, then we need some way to communicate the styling of token types to the editor. As discussed previously in Section 5.2.4, we used scope names instead.

The model for the AESI syntax service is shown in Figure 5.11. The service, represented by the `SyntaxService` interface, has the `getSyntaxInfo` method which takes a document URI and region within the document for which the editor needs the syntax information. It returns a `SyntaxInfo` object, which contains the tokens for at least the given region, in the order they appear in the document. It is permitted for the service to return tokens outside of the requested region, for example when the parser can only parse the whole file, or when a change in the document (such as adding a multi-line comment start) invalidated later parts of the document.

Each `Token` object describes a unique non-overlapping region of the source code, and assigns scope names to it. Gaps between tokens are allowed, they would receive the default style.

5.4 Code Completion

Our model for the code completion editor service is shown in Figure 5.13. The `getCompletionInfo` of the `CodeCompletionService` is given a caret location in a document and should either return nothing (when there are no proposals) or return a `CompletionInfo` object, which contains a list of completion proposals, and a *prefix* region. The `prefix` attribute specifies a region of the document that is highlighted by the editor as the prefix for the current list of completions, which is replaced when a proposal is accepted.

Each completion proposal, represented by the `CompletionProposal` interface, has at least `content`, which is the text or snippet inserted into the document when the user commits to the proposal. Before inserting a proposal, the prefix region specified by the `CompletionInfo` object is removed.

A code snippet can be used as the proposal's content (if the editor supports it, which is given by the `CodeCompletionConfiguration`'s `supportsSnippets` attribute), for which we use snippet syntax based on that of the TextMate editor, which incidently is extremely similar to

```
if (${1: some_variable} == null) {
    $0
    throw new IllegalArgumentException("$1 must not be null.")
}
```

Figure 5.12: Example code snippet, where `$1` is a placeholder whose value is substituted in two places, and which has a default value. The final caret location is denoted with `$0`.

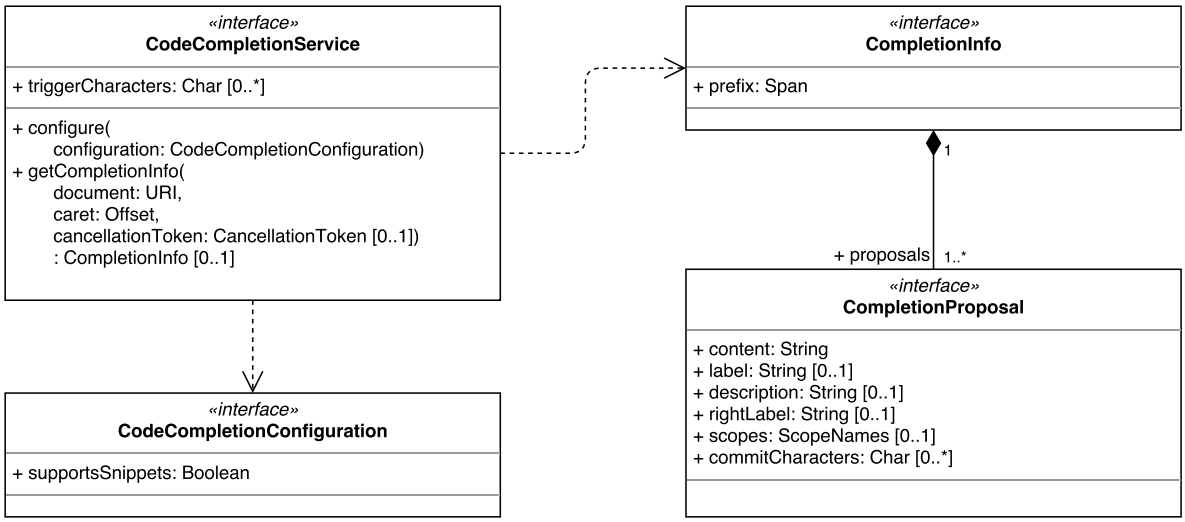


Figure 5.13: Overview of the AESI API for code completion.

the snippet syntax supported by editors such as VS Code, Sublime, Eclipse Che, and Atom. Figure 5.12 shows an example snippet using this syntax; the full definition is in Appendix D.

A proposal can have a custom label, for which the content is displayed when no label is specified. In many editors it can also show a short description text and right-aligned label. The right-aligned label is often used to show the (return) type of a proposal. While two of the editors support styling the label, we decided not to support this in AESI, as there is no single markup language or set of coloring and styling operations that are supported across editors.

Most editors can display an icon next to each proposal. The icon is determined by the editor adapter through the scope names applies to the proposal. For example, the `meta.class-.java` scope name might result in a Java class icon being displayed.

The characters that can be used to commit to a completion proposal depend on both the language and the proposal, and these are specified in the `commitCharacters` list. For example, in Java the user may commit to a member proposal by typing the member accessor operator dot (`.`), or commit to a method proposal by trying the opening parenthesis `(`. When the service returns only one proposal, some editors commit to that proposal immediately.

5.5 Structure Outline

The structure outline is a tree of declarations in the current document, such as classes and methods. In AESI each element in the tree is represented by the `StructureOutlineElement` interface, shown in Figure 5.14, which has a label and optionally an associated location in the source code. The icon and styling, if supported by the editor, are determined from the scope names of the element. For example, an editor may render an element with the scope

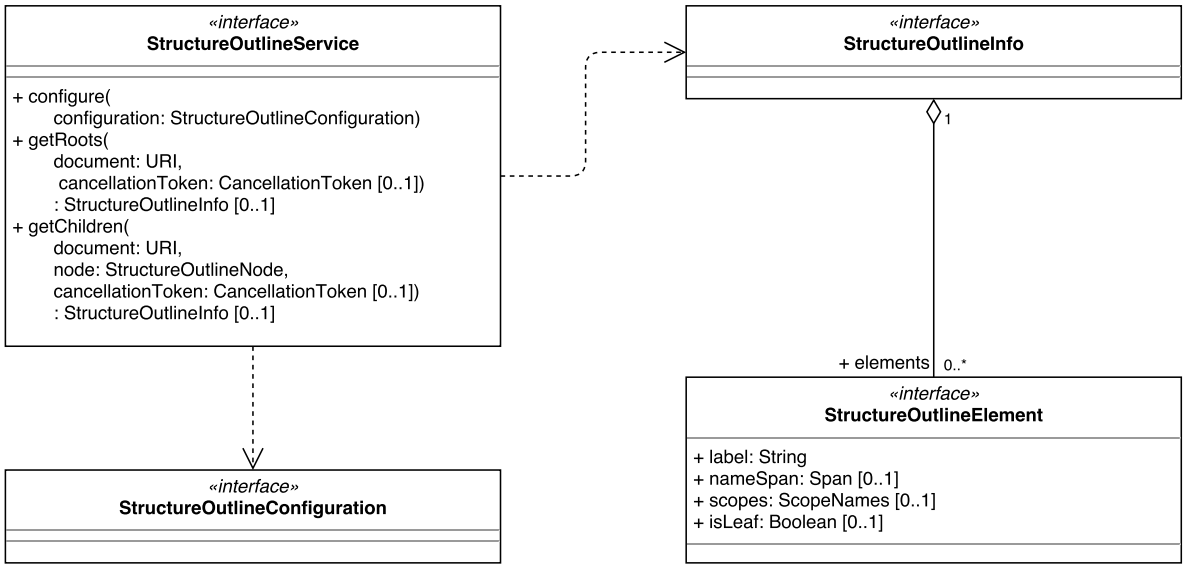


Figure 5.14: Overview of the AESI API for the structure outline.

names `meta.class.private, invalid.deprecated` with the icon of a class, a lock to signify that the class is private, and the name struck out to indicate that the member is deprecated.

For large files or deep structure outlines, it can take a significant amount of time to build the whole structure outline. Instead, we opted to allow the outline tree to be built lazily: the children of an element are only built and returned when the service is asked for them, for example when the user expands an element in the tree to reveal its children. To this end, the `StructureOutlineService` has a `getRoots` method that returns the root elements of the structure outline, and a `getChildren` method that returns the children of a particular element. To indicate to the editor whether it should show the *reveal children* button next to an element, we added the `isLeaf` attribute to the outline node. When its value is `true`, the editor knows the element has no children and it can hide the expansion button, without having to ask for the children of the element.

5.6 Reference Resolution

Our `ReferenceResolverService`, shown in Figure 5.15 returns the definitions of the reference at the given caret location in a document, and each definition specifies the span in a document where it occurs. While references usually have a single definition, multiple definitions are possible, such as when the definition is spread across files (for example, *partial* classes in C♯) or when the reference is ambiguous.

5.7 Code Actions

The code actions service provides the user with a menu of possible actions that can be applied at the caret location. The actions can range from adding an `import` statement for the current identifier to applying project-wide refactorings and inspections of the source code.

Code actions in our model in Figure 5.16 are provided by a `CodeActionService`, whose main method is given a location in a document and returns a list of possible actions. Each action has a label and command that is executed when the action is invoked. Optionally, the action has a span indicating the part of the source code to which the action pertains, and scope names which are used to determine the icon of the action.

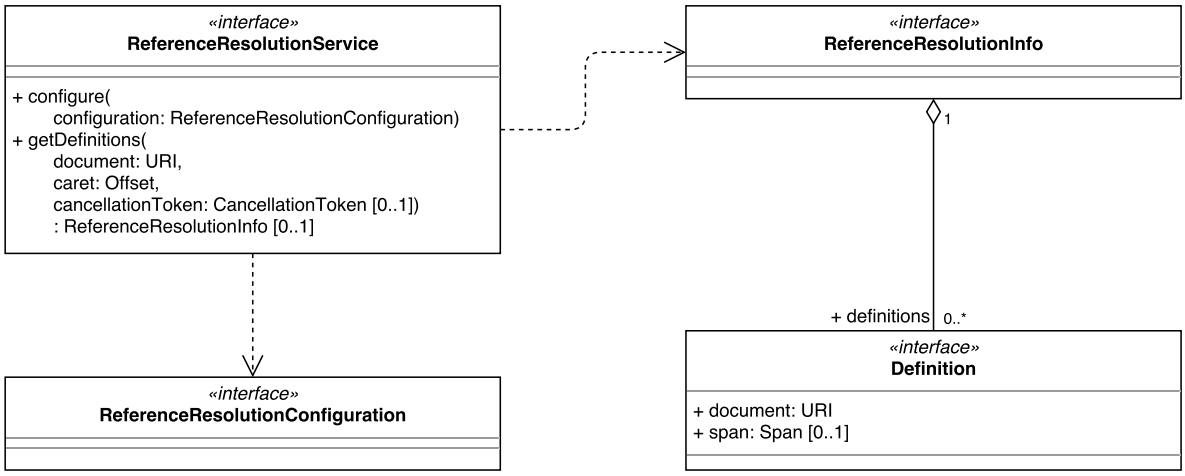


Figure 5.15: Model for resolving references to their definitions.

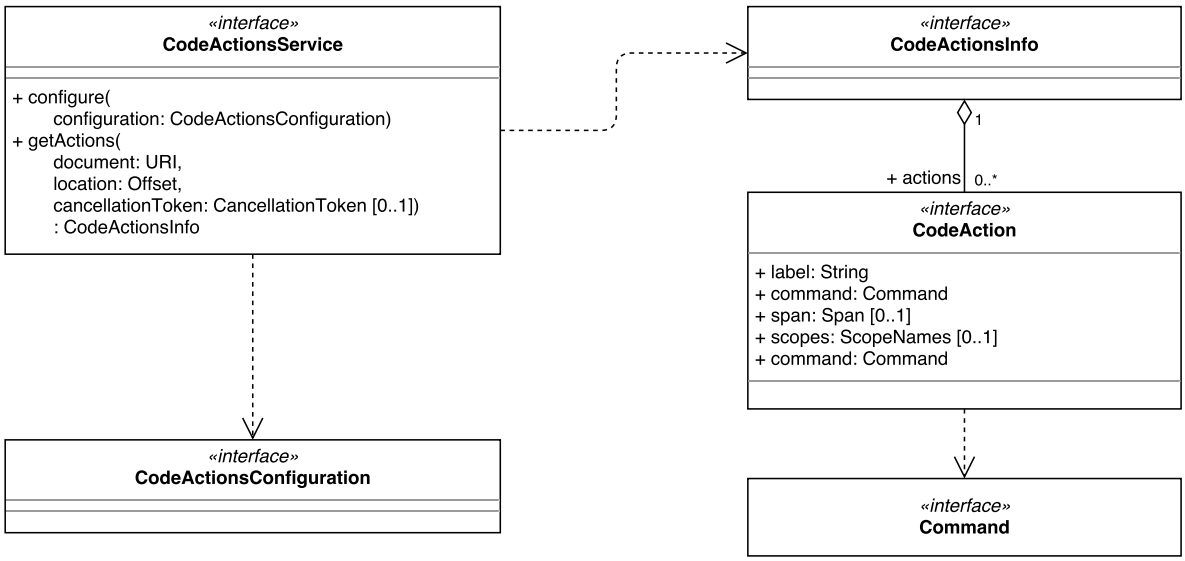


Figure 5.16: Model for code actions.

5.8 Debugging

A debugger allows the user to control the program's execution and inspect its state, such as the current point of execution, the values of local variables, and the call stack.

Debuggers in the various editors are very similar. They can launch a program, or attach to an already running program, and allow the user to set breakpoints, pause and resume execution, step to the next instruction, and step into or out of function calls. They display the state of the current program as a hierarchy of the current process, its threads, the thread's stack frames, and the local variables in a stack frame. Finally, most editors allow arbitrary expressions to be executed by the user. We will provide these features in our model as well.

The AESI model for debugging, shown in Figure 5.17, uses two services: the `DebugServerService`, provided by the language implementation, executes the debugging operations and provides the program's state; and the `DebugClientService`, provided by the editor adapter, handles debugger events. Whenever the user invokes a debugger operation, the corresponding method on the language's `DebugServerService` is called, such as `pause` when the user

wants to suspend program execution. The debug server will execute the operation at the earliest available opportunity and notify the editor of this fact through a call to one of the `DebugClientService` methods, for example `paused` once execution is actually suspended.

The debug server service has methods for starting, stopping, pausing and resuming execution, and stepping forward, into, and out of functions. The service can also set the current execution location somewhere else using the `goto` method, and return the current execution location through `getExecutionPoint`.

The current state is provided by our model through the `getProcess` method, which returns the current `Process`. The process has a list of threads, where each thread has a list of stack frames. The stack frames have local variables, each of which can have variables of themselves.

The debugger can execute arbitrary expressions through its `evaluate` method, which takes the expression as a string and returns a variable with the result of the expression. The debugger also supports breakpoints to be set in the code, where a breakpoint signals that execution must be paused when it is triggered. Optionally, a breakpoint can have a condition expression, where execution is only paused at a breakpoint when the condition is met.

While our debugger model is very featureful, editors still have debugger features that are not supported. Most notably, pausing and resuming individual threads is not supported, as the semantics of resuming only some threads in a program are not clear and not consistent across editors that support it. Another unsupported feature is *time-travel debugging*, where the debugger can step and execute backwards, undoing any previously executed instructions. Time-travel debugging has not been incorporated into our model, as there is currently only one editor (VS Code) that supports this.

5.9 Evaluation

To evaluate AESI, we implemented four editor services in three editors, and used them from two simple language implementations. This gives insight into whether AESI meets the requirements set previously in Section 5.1.

For evaluation we implemented the syntax coloring, structure outline, reference resolution, and code completion editor services, as these are the most supported services across popular editors, and also the four editor services supported by the Spoofax language workbench. The AESI language implementations and editor adapters can be found on GitHub at `https://github.com/Virtlink/aesi`.

We implemented AESI adapters for three editors: Eclipse, IntelliJ and VS Code. We chose Eclipse as it is the primary editor used by Spoofax, and chose IntelliJ as it is the editor whose model is most different from the other popular editors. Both Eclipse and IntelliJ are Java-based editors, so as the third editor we chose VS Code as it is an editor written in the JavaScript programming language, with built-in support for LSP. Having an AESI adapter for LSP could support other LSP-enabled editors such as Eclipse Che with very low effort, but we did not support these editors explicitly. Being able to adapt AESI for these three editors is a strong indicator that the AESI model is generic enough.

To test the language implementation side of AESI, we wrote two implementations in Java: one that uses the Pipelines for Interactive Environments (PIE) runtime for incremental editor services, and a dummy language implementation.

In the end, we ended up with six combinations of languages and editors, while writing only one implementation for each of the two languages and one adapter for each of the three editors. The effort of supporting more languages and editors will be even more reduced when new language implementations and editor adapters are created: just adding a single new language would already add three new language-editor combinations, as very little effort is required to support the three editors that now have adapters written for them.

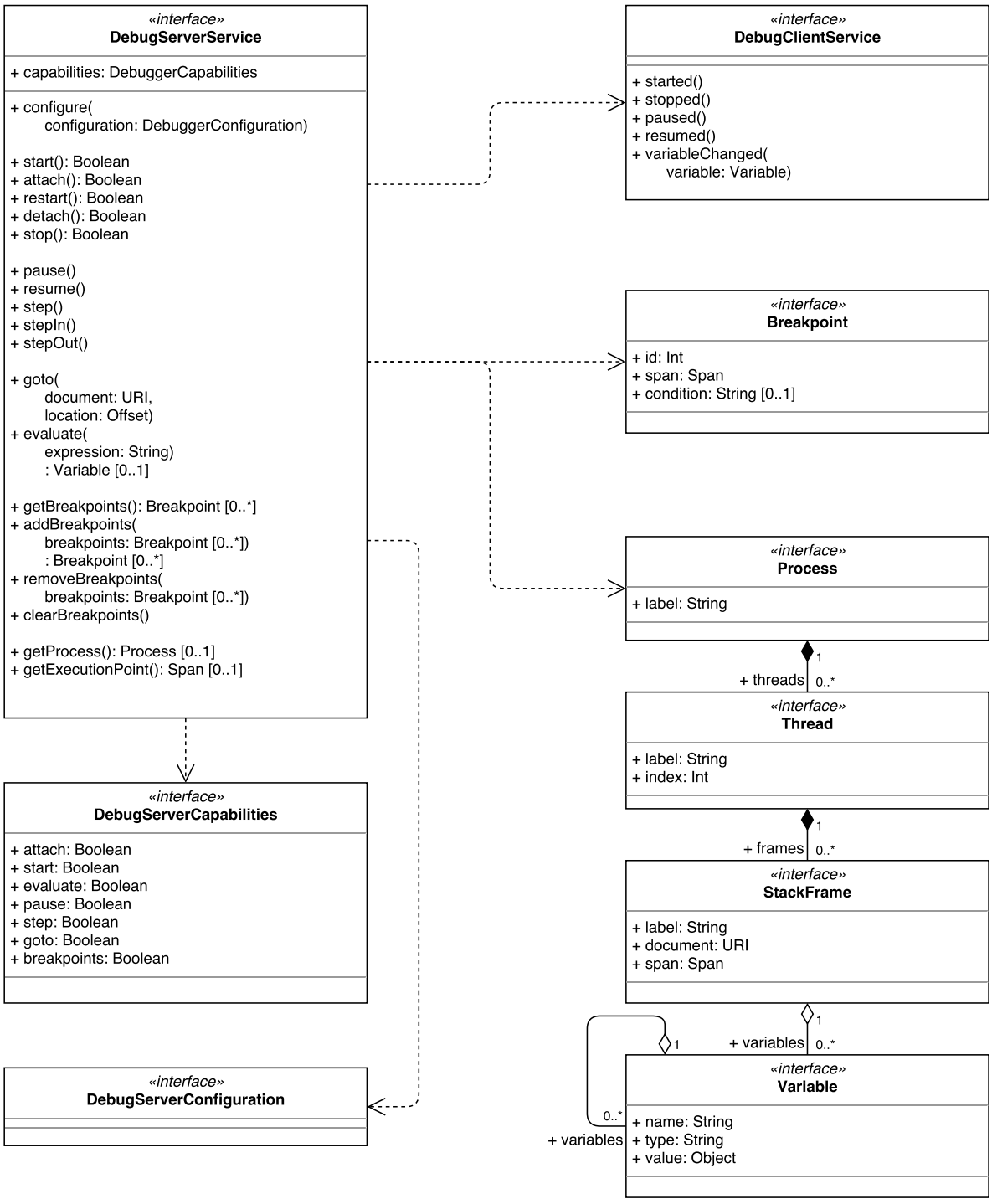


Figure 5.17: Model for debugging.

5.9.1 Editor Adapters

We implemented Java-based editor adapters for Eclipse, IntelliJ, and Language Server Protocol, which can be bound to any AESI language implementation written in Java.

Eclipse Editor Adapter

In the `aesi-eclipse` Eclipse editor adapter for AESI we have implemented support for four editor services: syntax coloring, code completion, reference resolution, and the structure outline.

Syntax Coloring Syntax coloring was implemented by having the editor schedule a *coloring job* every time something changed in the editor's document. The coloring job invokes the AESI `SyntaxColoringService` to get the tokens of the document and their associated scope names. The editor adapter uses the scope names to determine the appropriate styling for these tokens, and currently supports styling keywords, comments, entities, strings, variables, invalid, and deprecated tokens.

From the actual style and the span to which the token applies, a presentation object is created, which describes the styling of the source code words in the document. Finally, the new presentation is applied to the editor, replacing the previous presentation and updating the colors of the editor text.

Code Completion For code completion we implemented the `IContentAssistProcessor` interface of Eclipse. Its `computeCompletionProposals` method, which is called whenever the user invoked code completion, should return the list of completion proposals from which the user may pick.

To build the list of proposals, the content assist processor calls the `getCompletionInfo` method of the AESI `CodeCompletionService`, which returns the proposals. However, the completion proposals returned by the AESI service cannot used interchangeably with the proposals Eclipse expects, so a trivial mapping function takes each AESI proposal and creates the corresponding Eclipse proposal. The latter includes an icon, which is selected based on the scope names of the AESI proposal.

Reference Resolution As Eclipse does not have a dedicated API for reference resolution, we leveraged the hyperlink system instead. This editor service is used to detect and highlight hyperlinks in the source code, and allows custom code to be executed when the user click such a hyperlink.

Our implementation of Eclipse's `IHyperlinkDetector` interface calls upon the AESI `ReferenceResolutionService` to get the definitions at the current caret location. Each definition is mapped to an Eclipse hyperlink object, each of which has a label, type, and source location. When the user clicks a hyperlink, the focus of the editor is moved to the definition's location.

Structure Outline A structure outline in Eclipse must be implemented as a `ContentOutlinePage` with an associated *content provider* and *label provider*. Our content provider implementation calls the AESI `StructureOutlineService` to get the structure tree elements, and returns these objects to Eclipse to be displayed to the user. The editor calls upon our implementation of a label provider to get the label and icon for a given AESI tree element, where the icon is derived from the scope names of the element.

Conclusion While Eclipse plugin development in general is a bit cumbersome due to its reliance on OSGi, a platform for building modular and service-oriented applications that does not play well with modern build systems, the implementation of an AESI editor adapter for

Eclipse was straightforward. The APIs of Eclipse are sufficient to not need a lot of boilerplate code, without imposing strict requirements onto our implementation.

We created two Eclipse plugins: one for our dummy language implementation, and another for our PIE language implementation. These plugins, being merely projects that bind a language implementation to our editor adapter, have only three small classes in them: a dependency injection module that binds each AESI interface to its implementation, a plugin class that loads the dependency injection module, and an editor class that provides some dependencies to the Eclipse editor, as it does not support constructor injection. These plugins and their classes could easily be generated.

IntelliJ Editor Adapter

The AESI editor adapter for IntelliJ, `aesi-intellij`, has the following editor services: syntax coloring, code completion, reference resolution, and the structure outline. By default IntelliJ expects all editor services to work in terms of the ASTs of the source files, called Program Structure Index (PSI) structures in IntelliJ. As AESI works on the source file text instead, we had to either build these ASTs from the limited information returned by an AESI editor service, or circumvent them altogether.

Syntax Coloring The default way to implement syntax coloring in IntelliJ is to have the lexer and parser classes be generated from definitions written in JFlex and Backus–Naur form notation, and install these in a `ParserDefinition`, where the lexer and parser are invoked on the content of a document without any information on the filename and path of the document. While the AESI syntax coloring service does not require the filename and path, it is preferred to provide it when it is available. Therefore, we have instead overridden the `doParseContents` method of the element type that represents the language's source file, where we have access to the filename and path when we invoke the lexer and parser.

The lexer tokenizes the file according to the tokens returned by the AESI `SyntaxColoringService`. By representing tokens with different scopes by different IntelliJ PSI elements, we can determine the coloring they will have.

The parser would take the tokens and uses them to construct an AST. As we do not have any information as to what the AST should look like, we replaced the parser by an `AstBuilder` that builds a very shallow AST with all the tokens as leaves and direct children of the root node. This is sufficient for IntelliJ.

Code Completion Code completion implements the `CompletionContributor` interface of IntelliJ, whose `fillCompletionVariants` method fills a set with completion proposals. In our implementation the proposals are returned from a call to the AESI `CodeCompletionService` and then transformed to their corresponding IntelliJ *lookup elements*. All code completions are either represented by PSI elements, which we want to avoid as we have no proper AST, or as lookup elements, where each element describes the label, icon, styling, and other display attributes of a completion proposal.

Reference Resolution Reference resolution in IntelliJ is very much tied to the PSI elements. Therefore, we implemented the IntelliJ `PsiReferenceProvider` that returns an array of `PsiReference` objects when its `getReferencesByElement` is called with a given PSI element. Our implementation determines the text offset of the PSI element, uses that to call the AESI `ReferenceResolutionService`, and creates a `PsiReference` for each definition returned. A `PsiReference` refers to (part of) a PSI element, so we find the PSI element closest to the definition's offset and use that as the target element.

Structure Outline The structure outline editor service is an implementation of the IntelliJ `StructureViewModel`, which has methods for getting the root nodes and child nodes of the structure outline. While the nodes would normally be represented by PSI elements from the source document, we opted to create standalone nodes not tied to the source document. This gave us the freedom needed to convert the tree returned from a call to the `StructureOutlineService` to its corresponding IntelliJ structure outline view model, without the tree having to match the AST of the document. The downside is that clicking a tree element does not automatically move focus to the corresponding definition in the source code.

Conclusion Adapting AESI to IntelliJ is possible, but it is more cumbersome than adapting for any other editor, as IntelliJ heavily relies on its internal PSI representation of the source files. While we could have incorporated an AST as part of the AESI model, it would have unnecessarily burdened any language implementations. No matter how simple, language implementations would always have to provide a proper AST for their document, only because one of the editors required it. Instead, we believe that without an AST the other editor services can still be adapted to IntelliJ, perhaps with some extra effort.

We created two plugins for IntelliJ, one for the dummy language and another for PIE. Each plugin has a class with the dependency injection module, that binds the language implementation to the AESI interfaces, and a plugin class that loads the dependency injection module. As most of the classes in IntelliJ cannot support dependency injection through their constructor, we had to resort to extending these classes in the plugin and providing the dependencies manually. This resulted in an additional fifteen classes just for dependency injection, but these are very trivial and could easily be generated.

LSP Editor Adapter

For the LSP editor adapter `aesi-lsp`, we implemented a language server written in Java, which can respond to the reference resolution and code completion requests. We implemented only code completion and reference resolution, as syntax coloring and the structure outline are not part of LSP. While we tested our implementation only against VS Code, it should work with any LSP-enabled editor.

Virtual File System AESI specifies that the source of truth for the content of the documents lies with the editor, and that the editor services can access the content through the `ResourceService` provided by the editor adapter. For the LSP editor adapter, the resource service listens for any changes made to the documents in the editor, and merges them with a local snapshot of those documents. This allows the resource service to provide the latest version of every document to the editor service, without having to request the whole document content from the remote editor.

Code Completion Code completion is implemented by handling the LSP `completion` request. The implementation translates from the LSP document URI and line-character-based location to the AESI document URI and offset-based location, and calls the `CodeCompletionService`. The completion proposals it returns are translated back to LSP data structures, and sent back as the response.

Reference Resolution For reference resolution, we handle the `definition` request. The request specifies the document URI and line-character-based location for which to get the definitions. These are again translated to their corresponding AESI document URI and offset-based location, and passed to the `ReferenceResolutionService`. The resulting list of definitions is converted to an LSP-compatible representation and returned to the caller.

Conclusion The two editor services for the LSP editor adapter were simple to implement, as their structures resemble the AESI structures. Most complex was the resource service, that has to track all changes to the documents. We bundled the editor adapter with our two language implementations, the dummy language and the PIE implementation, into separate language servers that can be launched from the command-line or from an editor upon loading a language.

The projects that bundle the editor adapter with each language implementation have only one class and one method: the dependency injection module binding the interfaces to their implementations, and a `main` method that reads the command-line arguments and starts the language server. This boilerplate code can easily be generated.

Finally, VS Code requires that ever language has their own editor plugin, even those using an LSP language server. Therefore we created such a plugin for each of the two language implementations. These plugins are simple JavaScript projects, most of which is generated by the VS Code plugin code generator. The generated code has been changed to launch the language server when the plugin is loaded.

Two of the four editor services we implemented for the other editor adapters, syntax coloring and the structure outline, are not supported in LSP and therefore not implemented in our adapter. Both could be supported in specific editors by adding the editor-specific code to the editor plugin, and adding custom LSP requests to the protocol. For example, syntax coloring is by default only supported through declarative TextMate grammars, but a VS Code plugin could add a `coloring` request to the LSP protocol and use the response to color the source code in the editor.

5.9.2 Requirements

The requirements we set in Section 5.1 were:

1. Editor services portable across editors
2. Expose maximum features supported by more than one editor
3. Impose minimal requirements on language implementations
4. Design for use with a communication protocol without enforcing one

We will discuss each of these requirements, starting with the last one.

Communication Protocol

We have not specified a particular communication protocol to use between editor adapters and language implementations, but we still needed to take into account that AESI will be used over some protocol.

One way we designed for use with a communication protocol is by separating data from behavior. Only the services contain behavior (methods), and all other interfaces contain only data. This way, a service need not be an actual class instance, but instead can be, for example, a collection of remote procedure calls or endpoints in a web service.

Similarly, cycles are not allowed in the AESI data interfaces, such as a tree node with references to both its children and its parent node. The lack of cycles allows the data to be represented as nested data structures, such as in Extensible Markup Language (XML) or JavaScript Object Notation (JSON), without the need for references.

The editor adapter we implemented for VS Code uses LSP as the communication protocol between the JavaScript-based editor and the language implementations written in Java. Our LSP editor adapter starts a language server which can respond to requests from any LSP-enabled language client, such as the VS Code editor.

The separation between behavior and data in the AESI interfaces allowed our implementation of the LSP editor adapter to stay relatively simple: LSP data structures can be transformed to and from their corresponding AESI data structures, while the behavior of the editor services is modeled as requests in the language server.

Minimal Requirements

Editor services can start our simple, but as the programming language evolves, their editor services get more complex, perhaps even using data structures and techniques we have not conceived of yet. By imposing minimal requirements on language implementations, we aim to not restrict the implementations or force them into a particular model.

We provided two language implementations: a dummy implementation, and a PIE implementation. The dummy implementation just returns static dummy data for the editor services, and shows how simple the simplest implementation can be. There is no requirement to maintain any state, or provide complex data structures such as an AST.

PIE is a framework and domain-specific language (DSL) for writing interactive editor service pipelines. A pipeline is a graph of builders that have dependencies on one another. The PIE implementation calls the PIE builders whenever an editor service is invoked. A PIE builder can have an arbitrarily complex implementation, but AESI puts neither restrictions nor requirements on them.

Maximum Features

We can compare the features provided by AESI with the features provided by the various editor services in the most popular editors (Chapter 4). Where a feature is supported by an API in more than one editor, it is also directly or indirectly supported by AESI. Icons are an example of an indirectly supported feature, as they are determined by the editor adapter through the scope names specified on AESI objects.

Portable Editor Services

The fact that we can use our two language implementations with the three editors for which we have written editor adapters, is a strong indication that the AESI editor services are portable across editors.

Other Observations

Push-based Editor Services The AESI editor services are pull-based, that is, the editor performs a service request and waits for the response, then displays the results.

However, editor services that take a long time to execute, such as control and data flow analyses, are often executed in the background. Push-based services would push the results to the editor once they are available, and not wait for the editor to ask for them. For example, simple syntax-based coloring can return the coloring of the document near instantly, but semantic coloring, such as whether an identifier is deprecated, often needs a complex analysis across documents, after which the results can be pushed to the editor. Such push-based editor services are currently not supported by AESI, but will be part of PIE. Therefore, to fully support PIE, we would need to support push-based services as well.

One way in which push-based editor services can be supported, is to implement a system where the editor service can notify the editor that new results are available. The editor service, upon receiving the notification, can then pull the new results as normal.

Virtual File System The virtual file system, exposed by the *resource manager*, is sufficient for editor services to read from, but does not allow any mutation. Actions such as changing,

creating, deleting, or renaming a document are not supported by the resource manager. To change a document, editor services have to return `TextEdit` objects instead. Whether there is any benefit to allowing mutation in the AESI resource manager, or whether that would introduce unacceptable complexity in all editor adapters remains an open question.

Additionally, there is currently no way in AESI to see the extent of changes to the documents, as the resource service only returns whole documents. Editor services could use information about each change to perform incremental computations, reusing any previous results that are still valid for the unchanged parts of a document.

Chapter 6

Related work

Existing technologies can provide editor services for a programming language, some of which allow portable editor services across editors. In this chapter we will briefly explore these technologies and compare them with Adaptable Editor Services Interface (AESI). We discuss portability in general in Section 6.1, followed by portability through language workbenches in Section 6.2. Two other solutions to the integrated development environment (IDE) portability problem, Monto and Language Server Protocol (LSP), are discussed in Section 6.3 and Section 6.4 respectively.

6.1 General Portability

Portability in general is defined by Tanenbaum, Klint, and Böhm (1978) as "a measure of the ease with which a program can be transferred from one environment to another". They express this measure as the effort to port a program compared to reimplementing the program. Mooney (2004) makes this explicit and proposes the *degree of portability* as a metric for estimating whether software should be ported or redeveloped, which they define as:

$$DP = 1 - \frac{\text{cost of porting}}{\text{cost of redevelopment}}$$

Since we want to avoid redevelopment of our language's editor services for every code editor, we must bring the cost of porting down. Back in 1997, Mooney (1997) already identified high-level strategies that increase software portability. Relevant for our purposes of portable editor service implementations are: the *identification* of external interfaces, which in our case are the interfaces between the editor services and the code editors; the *isolation* of the parts of the software that need to be adapted, in our case, for each code editor; and finally a *portable mindset* that influences all choices, from the choice of programming language to the way documentation is written. While adhering to these practices by themselves do not cause editor services to be portable, they can help when designing for portability.

6.2 Portability using Language Workbenches

Language workbenches help with the development of programming languages and their editor services. Through a language definition, the workbench can derive fully-functional editor services for any language. However, most language workbenches, such as Meta Programming System (MPS)[1] and Rascal[2], support only one editor and are tightly integrated into it. In such workbenches languages are not portable across editors.

[1] `https://www.jetbrains.com/mps/`

[2] `http://www.rascal-mpl.org/`

The Spoofax[3] and Xtext[4] language workbenches, however, support more than one editor. Spoofax and Xtext languages can be used both in Eclipse and IntelliJ, and Xtext additionally supports browser-based editors. A Spoofax language definition, once built, is usable in both Eclipse and IntelliJ without modification, making the languages portable across supported editors. Xtext languages are not portable, as Xtext generates a separate plugin for each supported editor, and these plugins and their generators are not reusable in other editors. In both cases, the languages become very dependent on the language workbench themselves: a language definition has to be written from scratch for a particular workbench, and it is hard if not impossible to reuse existing service implementations.

6.3 Portability using Language Server Protocol

Language Server Protocol (LSP) (Microsoft 2018a) is a communication protocol between a server, exposing the language's editor services, and a client, the editor. It models many editor services such as code completion and reference resolution, but lacks essential services such as syntax coloring, structure outline, and debugging. Additionally, a language implementing LSP still needs an editor-specific plugin for every editor it wants to support. As these plugins contain, for example, the language's syntax definition, they cannot easily be generated, making LSP only a partial solution for portable editor services.

While LSP provides a wide selection of editor services, their choice of supported services reflect their support in VS Code, the main editor implementing LSP. For example, *code lens* is an editor service supported by Microsoft in their Visual Studio and VS Code editors, and therefore supported in LSP, whereas a structure outline is not supported by either Visual Studio or VS Code, and therefore not by LSP, even though most other editors support it.

Similarly, some of the LSP interfaces are noticeably tailored toward use in VS Code. For example, whenever an icon can be specified, it must be one of the pre-defined options for which icons are available in VS Code.

Other aspects of LSP are underspecified, such as the file system. LSP identifies files by a Uniform Resource Identifier (URI), but the protocol does not specify the format of the URI. Different editors can have completely different URI formats, yet language servers are expected to work with any editor. It is also unspecified how the language server can map any URI to its physical file, or how the language server is supposed to access these files. Additionally, language servers are implicitly assumed to run on the same system as the editor, to have access to the same local files, as there is no mechanism by which an editor and language server can run on different systems.

6.4 Portability using Monto

Keidel, Pfeiffer, and Erdweg (2016) proposed Monto as a solution to the *IDE portability problem* they identified, which has portable editor services. Monto is a language- and editor-agnostic intermediate representation and service system, that sits between language services and editor plugins.

Monto's architecture consists of a Monto plugin in the editor and a *broker* in a separate process with whom the plugin communicates using the ZeroMQ communication protocol (Sústrik 2012). The broker manages stateless *language services* that depend on files and each others products, such as abstract syntax trees (ASTs). The broker calls a language service and passes it the new files and products whenever a file or intermediate product changes. The service's results, such as a collection of errors and warnings, a structure outline, or the

[3] `http://www.spoofax.org/`

[4] `https://www.eclipse.org/Xtext/`

syntax coloring, are written in the Monto intermediate representation (IR) and sent back to the Monto plugin to be displayed to the user.

6.4.1 Stateless Services

Where LSP and AESI only describe the interface though which editors can use a language's editor services and have no architecture that aids in their implementation, Monto's architecture allows the services to be small by requiring that the services are stateless, and managing all the dependency tracking and caching in a common Monto broker. While this makes it easy to add additional language services to Monto, it also prohibits incremental processing of changes to the service dependencies, and makes it impossible to implement stateful editor services such as a debugger or a Read-Eval-Print Loop (REPL) interactive programming environment.

As AESI does not restrict the implementation of editor services, they can be stateful and arbitrarily complex. While the lack of supporting infrastructure increases the effort for implementing the service, it makes it easier to support incremental services, allow the implementation of interactive services such as a debugger, or to adapt existing tools to AESI, such as the RustFmt code formatter for the Rust language, or the Flow type checker for JavaScript.

The stateless nature of Monto services also makes it more difficult to deal with dynamic dependencies. For example, when a type checker service starts with analyzing the main source file, it also needs access to any files imported in that source file, and possibly any files imported in those imported files as well, until all the files in the project have been explored. In Monto, whenever a service needs additional files, it asks the broker to add these dependencies, and stops its execution. The broker then restarts the service, this time with the requested files added to its original dependencies. This cycle repeats until the service requests no more files and can run to completion. However, an AESI service can fetch its dynamic file dependencies from the *resource service* instead, and continue.

6.4.2 Separation of Concerns

Some aspects of the editor services are concerns of the language, whereas other aspects are editor concerns, and in some places Monto IR mixes these concerns, making it less editor- and language-agnostic.

The Monto IR prescribes an AST format to be used, where each AST nodes has a name, list of child nodes, and an associated location in the source code. The node names and their semantics cannot be standardized, as they are clearly language-dependent, making the AST unusable for editors that deal with an AST, of which IntelliJ is the only example we found. This makes the AST arguably a language concern that the protocol should not try to nail down, also because it precludes the use of a *desugared* AST where the nodes have no longer a relation with the source code, or a different AST format altogether, such as the ATerm annotated AST format (Brand et al. 2000). As ASTs are not part of the AESI model and unspecified, a language implementation is free to choose any AST format, or even to use none at all for a very simple language.

Monto specifies how, for example, the nodes in a structure outline tree can have associated icons, where an icon is specifies by its Uniform Resource Locator (URL), which the editor is expected to fetch and display. Some editors, such as VS Code, have no support for custom icons, and even for editors that do, the icons are not likely to match the general theme of icons in the editor. In the worst case the icons are unusable or confusing due to a lack of contrast with whatever light or dark user interface (UI) theme the user has configured.

Similarly, a Monto syntax coloring service has to return the exact colors, font, style, and size that the editor is expected to use to display the syntactic tokens of a source file. However, the editor may not support fonts, differently sized text, italic, or bold styles, or certain colors

(such as sixteen colors in a terminal or two colors on an e-ink display), or a font may not be installed on the system.

Even with editor support, the colors may clash with the editor's color scheme, such as when black text is used on a dark background. Keidel, Pfeiffer, and Erdweg recognize this problem as well, and solve this through adding a configuration system to Monto. The language services, such as the syntax coloring service, publish their configuration options to the editor, which are displayed to the user by the editor's Monto plugin, and any changes to the configuration are sent back to the language service. The user will have to change both the editor's own color scheme settings and the Monto services color scheme, only to have the syntax service repeat the exact same colors back to the editor for every token.

Arguably, icons and the exact syntax coloring are editor concerns, and a language should not have to deal with them. In AESI, this is solved by letting the editor determine the icon and syntax color based upon a *scope name*, which is a standardized but extensible identifier provided by the language implementation. This also removes most of the configuration from the AESI editor services.

Chapter 7

Conclusion

Portable editor services allow a language implementation to provide its editor services across editors with minimal effort, giving developers the freedom to choose their preferred editor. Therefore, the main question of this thesis was: how can we achieve portable editor services?

Language definitions created in the Spoofax language workbench automatically gain support for their editor services in supported editors. We used Spoofax Core, the platform-independent core of the Spoofax language workbench, as an opportunity to answer the question whether Spoofax Core is truly platform-agnostic, by porting Spoofax Core to another platform next to the already supported Eclipse integrated development environment (IDE). This automatically provides editor services for languages defined in Spoofax, making these languages portable with no extra effort.

Our first contribution is an implementation of Spoofax Core for the IntelliJ IDE. This shows that Spoofax Core is platform-agnostic, because the IntelliJ architecture is very different from that of Eclipse. As language definitions written in Spoofax automatically provide editor services for any editor supported by Spoofax, the mere implementation of Spoofax Core for IntelliJ adds IntelliJ support to all current and future Spoofax languages. Spoofax languages are thus extremely portable.

However, as Spoofax Core only works with languages written using the Spoofax language workbench, the editor services are not truly portable across languages in general. Additionally, we found that as Spoofax evolves, updating the IntelliJ implementation and fixing its bugs are easy to neglect, especially when most people are using the more mature Eclipse IDE.

Therefore, as part of our main question, we asked how we can generalize the editor services such that even big changes to their implementation requires little to no change in their ports. To answer this, we explored the editor services of ten popular editors. Our second contribution is an overview of the application programming interface (API) through which these editor services can be implemented in these editors, serving as a starting point for actual implementations and a guideline for a generic editor services interface.

We found that while the implementation languages, features, and API-specific details of the editors differ, their overall architecture is very similar. The biggest outlier is IntelliJ, as it relies on a language building an abstract syntax tree (AST) representation on top of which many of the editor services are built.

Our third contribution is the generalization of the various editor APIs into set of generic editor services interfaces that are easily adaptable to various editors, called the Adaptable Editor Services Interface (AESI). It can also be used to decouple Spoofax Core from its editor implementations, severely reducing the maintenance effort of the editors Spoofax supports. We evaluated AESI by implementing four editor services (syntax coloring, code completion, reference resolution, and the structure outline) in editor adapters for three different editors (Eclipse, IntelliJ, and VS Code), and creating two different implementations. This shows that

AESI can be adapted to various editors, including the deviant API of IntelliJ.

Our overall conclusion is that well designed generic interfaces for editor services are the primary means to achieving portable editor services, and that details such as the communication protocol between editor services and editors are not as important, as they are relatively easy to adapt. AESI provides such a generic set of editor services interfaces for portable editor services, answering the main question of this thesis.

7.1 Future Work

An implementation of AESI for Spoofax Core would test the extensibility of AESI, as it does not currently support editor services such as dynamic language loading, and would allow Spoofax languages to be used in these editors with minimal effort. It would also reduce the effort of maintaining the various editors while Spoofax evolves, as the interfaces should remain stable and their implementations should only change in response to changes in the editors themselves.

To further validate AESI and facilitate its adoption, the editor adapters for IntelliJ, Eclipse, and VS Code should be completed by implementing the remaining editor services. AESI is, of course, not limited to existing desktop-based editors, and it would therefore be possible to build a web-based editor that takes advantage of the AESI editor services.

The AESI debugger is the most complex editor service that we have not been able to evaluate with an implementation. It would be interesting to see how easily the debugger model can be adapted to work with various IDEs, and how feature complete it is by implementing it in a language or language workbench such as Spoofax.

Bibliography

Anderson, Wade (2017). *VS Code Extensions using CodeLens*. URL: https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup (visited on 02/16/2018).

Aniyan, Mathew (2012). *Writing a Visual Studio 2012 Unit Test Adapter*. URL: https://blogs.msdn.microsoft.com/visualstudioalm/2012/07/31/writing-a-visual-studio-2012-unit-test-adapter/ (visited on 02/16/2018).

Apache (2015). *Maven: Project Dependencies*. URL: https://maven.apache.org/ref/3.3.3/maven-core/dependencies.html (visited on 02/16/2018).

Apple (2010a). *Old-Style ASCII Property Lists*. URL: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/PropertyLists/OldStylePlists/OldStylePLists.html (visited on 02/16/2018).

— (2010b). *Understanding XML Property Lists*. URL: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/PropertyLists/UnderstandXMLPlist/UnderstandXMLPlist.html (visited on 02/16/2018).

Atom (2015). *Atom Issue: foldingStartMarker and foldingStopMarker ignored*. URL: https://github.com/atom/first-mate/issues/48 (visited on 02/16/2018).

Barr, Earl T. et al. (2016). "Time-travel debugging for JavaScript/Node.js". In: *FSE*, pp. 1003–1007. DOI: http://doi.acm.org/10.1145/2950290.2983933.

Ben-Kiki, Oren (2009). *YAML Ain't Markup Language Version 1.2*. URL: http://www.yaml.org/spec/1.2/spec.html (visited on 02/16/2018).

Brand, Mark G. J. van den et al. (2000). "Efficient annotated terms". In: *SPE* 30.3, pp. 259–291.

Breslau, Dan (2009a). *Eclipse: Rich Hovers Redux*. URL: http://www.outofwhatbox.com/blog/2009/05/eclipse-rich-hovers-redux/ (visited on 02/16/2018).

— (2009b). *Using the new Rich Text Hovers in Eclipse 3.4*. URL: http://www.outofwhatbox.com/blog/2009/04/using-the-new-rich-text-hovers-in-eclipse-34/ (visited on 02/16/2018).

Brooks, Evan (2014). *Coding in color: How to make syntax highlighting more useful*. URL: https://medium.com/@evnbr/coding-in-color-3a6db2743a1e (visited on 02/16/2018).

Charles, Philippe, Robert M. Fuhrer, and Stanley M. Sutton Jr. (2007). "IMP: a meta-tooling platform for creating language-specific ides in eclipse". In: *ASE*, pp. 485–488. DOI: http://doi.acm.org/10.1145/1321631.1321715.

Cloud9 (2018a). *Cloud9 SDK: Builders*. URL: https://cloud9-sdk.readme.io/docs/builders (visited on 02/16/2018).

— (2018b). *Cloud9 SDK: Code Folding*. URL: https://cloud9-sdk.readme.io/docs/code-folding (visited on 02/16/2018).

— (2018c). *Cloud9 SDK: Debugger Plugin*. URL: https://cloud9-sdk.readme.io/docs/debugger-plugin (visited on 02/16/2018).

— (2018d). *Cloud9 SDK: Language Handlers*. URL: https://cloud9-sdk.readme.io/docs/language-handlers (visited on 02/16/2018).

Cloud9 (2018e). *Cloud9 SDK: Runners*. URL: https://cloud9-sdk.readme.io/docs/runners (visited on 02/16/2018).

Deva, Prashant (2005). *Folding in Eclipse Text Editors*. URL: https://eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html (visited on 02/16/2018).

Eclipse (2017a). *Eclipse Guide: Syntax coloring*. URL: https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Feditors_highlighting.htm (visited on 02/16/2018).

— (2017b). *Orion: How Tos: Code Edit*. URL: https://wiki.eclipse.org/Orion/How_Tos/Code_Edit (visited on 02/16/2018).

— (2018a). *Eclipse Guide: Contributing marker resolution*. URL: https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FwrkAdv_markerresolution.htm (visited on 02/16/2018).

— (2018b). *Eclipse Guide: Resource markers*. URL: https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm (visited on 02/16/2018).

— (2018c). *Eclipse Wiki: How do I add Content Assist to my language editor?* URL: https://wiki.eclipse.org/FAQ_How_do_I_add_Content_Assist_to_my_language_editor%3F (visited on 02/16/2018).

— (2018d). *Eclipse Wiki: How do I create an Outline view for my own language editor?* URL: https://wiki.eclipse.org/FAQ_How_do_I_create_an_Outline_view_for_my_own_language_editor (visited on 02/16/2018).

— (2018e). *Eclipse Wiki: How do I participate in a refactoring?* URL: https://wiki.eclipse.org/FAQ_How_do_I_participate_in_a_refactoring%3F (visited on 02/16/2018).

— (2018f). *Eclipse Wiki: How to decorate a TableViewer or TreeViewer with Columns?* URL: https://wiki.eclipse.org/FAQ_How_to_decorate_a_TableViewer_or_TreeViewer_with_Columns%3F (visited on 02/16/2018).

— (2018g). *Platform debug model*. URL: https://help.eclipse.org/mars/topic/org.eclipse.platform.doc.isv/guide/debug_model.htm (visited on 02/16/2018).

Eclipse Che (2016). *Eclipse Che Issue: Enhance Java debugger functionality*. URL: https://github.com/eclipse/che/issues/2611 (visited on 02/16/2018).

— (2017a). *Eclipse Che: Code Editors*. URL: https://www.eclipse.org/che/docs/assemblies/sdk-code-editors/index.html (visited on 02/16/2018).

— (2017b). *Eclipse Che: Commands*. URL: https://eclipse.org/che/docs/ide/commands/index.html (visited on 02/16/2018).

Efftinge, Sven (2016). *Language Server Protocol — Proposal for (Semantic) Coloring*. URL: https://github.com/Microsoft/language-server-protocol/pull/124 (visited on 02/16/2018).

Efftinge, Sven and Markus Völter (2006). "oAW xText: A framework for textual DSLs". In: *Workshop on Modeling Symposium at Eclipse Summit*.

Erdweg, Sebastian et al. (2013). "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge". In: *SLE*, pp. 197–217. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_11.

Eysholdt, M. and H. Behrens (2010). "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307–309.

Feldman, Stuart I. and W. Morven Gentleman (1990). "Portability - A No Longer Solved Problem". In: *csys* 3.2, pp. 359–380.

Fowler, Martin (2005). *Language Workbenches: The Killer-App for Domain Specific Languages?* DOI: http://www.martinfowler.com/articles/languageWorkbench.html.

Glozic, Dejan (2001). *Mark My Words*. URL: https://www.eclipse.org/articles/Article-Mark%20My%20Words/mark-my-words.html (visited on 02/16/2018).

Hiebert, Darren (2017). *Ctags*. URL: http://ctags.sourceforge.net/ctags.html (visited on 08/16/2017).

JetBrains (2015a). *IntelliJ Platform Support: Debugger for custom language plugin*. URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/206106219-Debugger-for-custom-language-plugin (visited on 02/16/2018).
— (2015b). *IntelliJ Platform Support: Graphical integration of running tests in plugin*. URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/206103879-Graphical-integration-of-running-tests-in-plugin (visited on 02/16/2018).
— (2015c). *IntelliJ Platform Support: Graphical integration of running tests in plugin*. URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/206103879-Graphical-integration-of-running-tests-in-plugin (visited on 02/16/2018).
— (2017). *Build Script Interaction with TeamCity*. URL: https://confluence.jetbrains.com/display/TCD10/Build+Script+Interaction+with+TeamCity (visited on 02/16/2018).
— (2018a). *IntelliJ Platform SDK: Additional Minor Features*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/additional_minor_features.html (visited on 02/16/2018).
— (2018b). *IntelliJ Platform SDK: Annotator*. URL: http://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/annotator.html (visited on 02/16/2018).
— (2018c). *IntelliJ Platform SDK: Code Completion*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/code_completion.html (visited on 02/16/2018).
— (2018d). *IntelliJ Platform SDK: Code Formatter*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/code_formatting.html (visited on 02/16/2018).
— (2018e). *IntelliJ Platform SDK: Documentation*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/documentation.html (visited on 02/16/2018).
— (2018f). *IntelliJ Platform SDK: Find Usages*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/find_usages.html (visited on 02/16/2018).
— (2018g). *IntelliJ Platform SDK: Formatter*. URL: http://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/formatter.html (visited on 02/16/2018).
— (2018h). *IntelliJ Platform SDK: Program Structure Index (PSI)*. URL: https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html (visited on 02/16/2018).
— (2018i). *IntelliJ Platform SDK: Quick Fix*. URL: http://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/quick_fix.html (visited on 02/16/2018).
— (2018j). *IntelliJ Platform SDK: References and Resolve*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/references_and_resolve.html (visited on 02/16/2018).
— (2018k). *IntelliJ Platform SDK: Rename Refactoring*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/rename_refactoring.html (visited on 02/16/2018).
— (2018l). *IntelliJ Platform SDK: Run Configuration Management*. URL: http://www.jetbrains.org/intellij/sdk/docs/basics/run_configurations/run_configuration_management.html (visited on 02/16/2018).
— (2018m). *IntelliJ Platform SDK: Safe Delete Refactoring*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/safe_delete_refactoring.html (visited on 02/16/2018).
— (2018n). *IntelliJ Platform SDK: Structure View*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/structure_view.html (visited on 02/16/2018).
— (2018o). *IntelliJ Platform SDK: Syntax Highlighting and Error Highlighting*. URL: http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/syntax_highlighting_and_error_highlighting.html (visited on 02/16/2018).

JetBrains (2018p). *IntelliJ Platform Walkthrough: Completion Contributor*. URL: http://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/completion_contributor.html (visited on 02/16/2018).

— (2018q). *IntelliJ Platform Walkthrough: Folding Builder*. URL: http://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/folding_builder.html (visited on 02/16/2018).

Kats, Lennart C. L. and Eelco Visser (2010). "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *OOPSLA*, pp. 444–463. DOI: 10.1145/1869459.1869497.

Keidel, Sven, Wulf Pfeiffer, and Sebastian Erdweg (2016). "The IDE portability problem and its solution in Monto". In: *SLE*, pp. 152–162. DOI: http://dl.acm.org/citation.cfm?id=2997368.

Kinable, Jan (2010). *Writing to the VS ErrorList*. URL: https://vsxexperience.net/2010/03/23/writing-to-the-vs-errorlist/ (visited on 02/16/2018).

Krishnamurthi, Shriram, Benjamin S. Lerner, and Joe Gibbs Politz (2017). *Programming and Programming Languages*. URL: http://papl.cs.brown.edu/2017/ (visited on 02/16/2018).

Larres, Jan (2017). *Tagbar*. URL: https://majutsushi.github.io/tagbar/ (visited on 02/16/2018).

Microsoft (2016a). *Formatting the Output of a Custom Build Step or Build Event*. URL: https://docs.microsoft.com/en-us/cpp/ide/formatting-the-output-of-a-custom-build-step-or-build-event (visited on 02/16/2018).

— (2016b). *Visual Studio Debugger Extensibility*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/debugger/visual-studio-debugger-extensibility (visited on 02/16/2018).

— (2016c). *Visual Studio SDK Walkthrough: Displaying Light Bulb Suggestions*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/walkthrough-displaying-light-bulb-suggestions (visited on 02/16/2018).

— (2016d). *Visual Studio SDK Walkthrough: Displaying QuickInfo Tooltips*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/walkthrough-displaying-quickinfo-tooltips (visited on 02/16/2018).

— (2016e). *Visual Studio SDK Walkthrough: Displaying Signature Help*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/walkthrough-displaying-signature-help (visited on 02/16/2018).

— (2016f). *Visual Studio SDK Walkthrough: Displaying Statement Completion*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/walkthrough-displaying-statement-completion (visited on 02/16/2018).

— (2016g). *Visual Studio SDK Walkthrough: Outlining*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/walkthrough-outlining (visited on 02/16/2018).

— (2016h). *Visual Studio SDK: Command Implementation*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/internals/command-implementation (visited on 02/16/2018).

— (2016i). *Visual Studio SDK: Inside the Editor*. URL: https://docs.microsoft.com/en-us/visualstudio/extensibility/inside-the-editor (visited on 02/16/2018).

— (2016j). *VS Code GitHub Issue: Language-aware folding*. URL: https://github.com/Microsoft/vscode/issues/3422 (visited on 02/16/2018).

— (2016k). *VS Code GitHub Issue: Symbols tree view*. URL: https://github.com/Microsoft/vscode/issues/5605 (visited on 02/16/2018).

— (2018a). *Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/specification (visited on 02/16/2018).

— (2018b). *The VS Code Debug Protocol*. URL: https://code.visualstudio.com/docs/extensionAPI/api-debugging (visited on 02/16/2018).

— (2018c). *VS Code Namespace API*. URL: https://code.visualstudio.com/docs/extensionAPI/vscode-api (visited on 02/16/2018).

— (2018d). *VS Code Tasks*. URL: https://code.visualstudio.com/docs/editor/tasks (visited on 02/16/2018).
— (2018e). *VS Code: Language Extension Guidelines*. URL: https://code.visualstudio.com/docs/extensionAPI/language-support (visited on 02/16/2018).
Moolenaar, Bram (2011). *Vim documentation: syntax*. URL: http://vimdoc.sourceforge.net/htmldoc/syntax.html (visited on 08/10/2017).
Mooney, James D. (1997). "Bringing portability to the software process". In: *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*.
— (2004). "Developing Portable Software". In: *ifip*, pp. 55–84.
Nolden, David (2009). *C++ IDE Evolution: From Syntax Highlighting to Semantic Highlighting*. URL: https://zwabel.wordpress.com/2009/01/08/c-ide-evolution-from-syntax-highlighting-to-semantic-highlighting/ (visited on 02/16/2018).
Notepad++ (2009). *Notepad++ And Scintilla*. URL: http://docs.notepad-plus-plus.org/index.php/Notepad%2B%2B_And_Scintilla (visited on 02/16/2018).
— (2014). *Auto Completion*. URL: http://docs.notepad-plus-plus.org/index.php/Auto_Completion (visited on 02/16/2018).
Oehme, Stefan (2015). *Xtext for IntelliJ - A first Beta*. URL: https://oehme.github.io/2015/05/22/xtext-intellij-beta.html (visited on 08/01/2017).
Ogarkov, Alexey (2017). *2.11 release of plugin for IDEA?* URL: https://github.com/eclipse/xtext-idea/issues/38 (visited on 02/16/2018).
Parr, Terence John and Russell W. Quong (1995). "ANTLR: A Predicated- :::: LL(k) :::: Parser Generator". In: *SPE* 25.7, pp. 789–810.
Scintilla (2018a). *Scintilla Documentation*. URL: http://www.scintilla.org/ScintillaDoc.html (visited on 02/16/2018).
— (2018b). *Scintilla Style Metadata*. URL: http://www.scintilla.org/StyleMetadata.html (visited on 02/16/2018).
Slant (2018). *What are the best cloud IDEs*. URL: https://www.slant.co/topics/713/~best-cloud-ides (visited on 02/16/2018).
Sloane, Anthony M. et al. (2014). "Monto: A Disintegrated Development Environment". In: *SLE*, pp. 211–220. DOI: http://dx.doi.org/10.1007/978-3-319-11245-9_12.
Stack Overflow (2017). *Stack Overflow Developer Survey Results 2017*. URL: https://stackoverflow.com/insights/survey/2017 (visited on 02/16/2018).
Sublime Text (2013). *Sublime Text Issue: Code Folding should be based on syntax, not on indentation level*. URL: https://github.com/SublimeTextIssues/Core/issues/101 (visited on 02/16/2018).
— (2018a). *Sublime Text API*. URL: http://docs.sublimetext.info/en/latest/reference/api.html (visited on 02/16/2018).
— (2018b). *Sublime Text: Build Systems*. URL: http://docs.sublimetext.info/en/latest/reference/build_systems.html (visited on 02/16/2018).
— (2018c). *Sublime Text: Symbols*. URL: http://docs.sublimetext.info/en/latest/reference/symbols.html (visited on 02/16/2018).
Sústrik, Martin (2012). *ZeroMQ*. URL: http://aosabook.org/en/zeromq.html (visited on 02/16/2018).
Szurszewski, Joe (2003). *We Have Lift-off: The Launching Framework in Eclipse*. URL: https://eclipse.org/articles/Article-Launch-Framework/launch.html (visited on 02/16/2018).
Tanenbaum, Andrew S., Paul Klint, and A. P. Wim Böhm (1978). "Guidelines for Software Portability". In: *SPE* 8.6, pp. 681–698.
TextMate (2018). *Manual: Navigation/Overview*. URL: https://manual.macromates.com/en/navigation_overview (visited on 02/16/2018).
The Open Group (1997). *Universal Unique Identifier*. URL: http://pubs.opengroup.org/onlinepubs/9629399/apdxa.htm (visited on 02/16/2018).

Toporov, Eugene (2013). *IntelliJ IDEA is the base for Android Studio, the new IDE for Android developers*. URL: https://blog.jetbrains.com/blog/2013/05/15/intellij-idea-is-the-base-for-android-studio-the-new-ide-for-android-developers/ (visited on 02/16/2018).

Vim (2011). *Vim script*. URL: http://vimdoc.sourceforge.net/htmldoc/usr_41.html (visited on 05/04/2017).

— (2017). *'insert' documentation*. URL: http://vimdoc.sourceforge.net/htmldoc/insert.html (visited on 04/11/2017).

Vim Tips Wiki (2017). *Creating your own syntax files*. URL: http://vim.wikia.com/wiki/Creating_your_own_syntax_files (visited on 02/16/2018).

Visser, Eelco (2015). *Declare your Language*. URL: https://github.com/MetaBorgCube/declare-your-language (visited on 02/16/2018).

Wang, Hanson (2017). *Introducing Atom IDE UI*. URL: https://nuclide.io/blog/2017/09/12/Introducing-Atom-IDE-UI/ (visited on 09/12/2017).

Wright, Darin and Bjorn Freeman-Benson (2004). *How to write an Eclipse debugger*. URL: https://eclipse.org/articles/Article-Debugger/how-to.html (visited on 02/16/2018).

xndcn (2018). *Atom Ctags Package*. URL: https://atom.io/packages/symbols-tree-view (visited on 02/16/2018).

Acronyms

AESI Adaptable Editor Services Interface

ANTLR Another Tool for Language Recognition

API application programming interface

ASCII American Standard Code for Information Interchange

AST abstract syntax tree

BNF Backus–Naur form

CSON CoffeeScript Object Notation

CSS Cascading Style Sheets

DSL domain-specific language

FTP File Transfer Protocol

HTML HyperText Markup Language

IDE integrated development environment

IMP Eclipse IDE Meta-tooling Platform

IO input/output

IR intermediate representation

JAR Java Archive

JPS JetBrains Project System

JSON JavaScript Object Notation

JVM Java Virtual Machine

LL Left-to-right Leftmost derivation

LR Left-to-right Rightmost derivation

LSP Language Server Protocol

MPS Meta Programming System

OSGi Open Services Gateway initiative

PAPLJ Programming and Programming Languages in Java

PEG parsing expression grammar

PIE Pipelines for Interactive Environments

plist Property List

PSI Program Structure Index

REPL Read-Eval-Print Loop

RPC remote procedure call

SDF Syntax Definition Formalism

SGLR Scannerless Generalized LR parser

SPT Spoofax Testing language

SQL Structured Query Language

UI user interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

UUID Universally Unique Identifier

YAML YAML Ain't Markup Language

XML Extensible Markup Language

Appendix A

Comparing Editor Services

In Chapter 4 we explored how the six most interesting editor services can be implemented for the most popular editors. These editor services were syntax coloring, code completion, structure outline, reference resolution, code actions, and debugging. The remaining editor services are covered in Appendices A.1 to A.7: code folding, hover documentation, signature help, automatic formatting, rename refactoring, diagnostic messages, and testing.

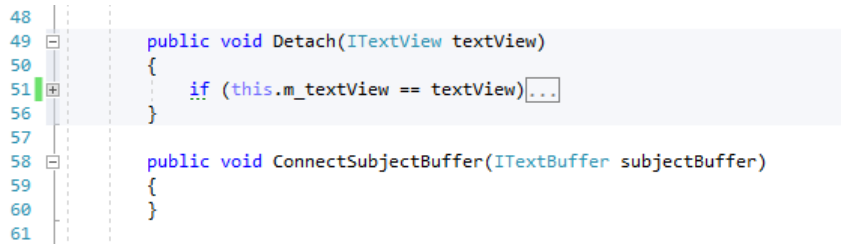


Figure A.1: Code folding in Visual Studio. A collapsed block of code, shown as ellipses, and an expanded block highlighted. The left margin indicates the state and extent of each block.

A.1 Code Folding

Code folding is the ability to collapse and expand regions of the source code, as shown in Figure A.1. Examples of collapsible blocks include classes, methods, parameter lists, blocks of statements, multi-line comments, and regions delimited by special comments (`//region` in Java) or preprocessor directives (`#region` in C♯).

A.1.1 Editor Support

A dedicated application programming interface (API) for code folding is provided by only five of the editors: Visual Studio, Eclipse, IntelliJ, Eclipse Che and Notepad++. In the editors VS Code, Sublime, and Atom the code folding is based on heuristics, such as the indentation level of the source code. These editors do not support language-aware folding regions, and supporting this in the future are currently open issues. (Microsoft 2016j; Sublime Text 2013; Atom 2015)

For both Vim and Cloud9 foldable regions are defined in the syntax definitions. Regions in Vim's syntax definitions (see Section 4.1.2) can specify whether they are foldable (Vim Tips Wiki 2017). Regular expressions are used to specify the boundaries of foldable regions in a Cloud9 grammar (Cloud9 2018b). These expressions are similar to those found in TextMate grammars.

Visual Studio

Visual Studio relies heavily on a system of *tags*, which allow language plugins to designate regions of code for special treatment. Foldable regions of any type (both lexical regions and uses-defined regions) are specified by tagging them with an *outlining region tag* that specifies the collapsible region of source text.

From the API, shown in Figure A.2, we can deduce that an tagged region can specify whether it is collapsed by default, and whether it is designated to be an *implementation region*, such as the body of a function, for which Visual Studio has an option to collapse only those, leaving the definitions visible. The `collapsedForm` attribute provides the object to display in place of the collapsed region, such as a string or a user interface (UI) element. When the user hovers their mouse over the collapsed region, `collapsedHintForm` determines the content of the tooltip that appears. (Microsoft 2016g; Microsoft 2016i)

Notepad++

Notepad++ uses the Scintilla editor component, and it allows plugins to specify code folding by assigning indentation levels to lines of code. The relevant API is shown in Figure A.3.

Implementations of Scintilla's `ILexer4` interface can specify the code folding by implementing the `fold` method. When given a document and region within that document, the

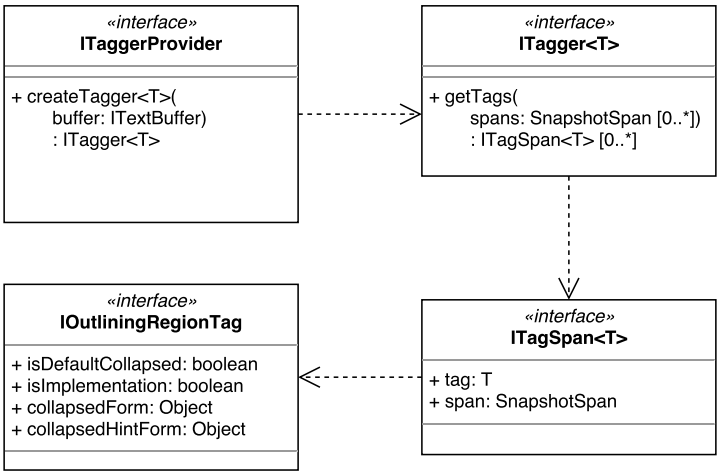


Figure A.2: Overview of the API for code folding in Visual Studio.

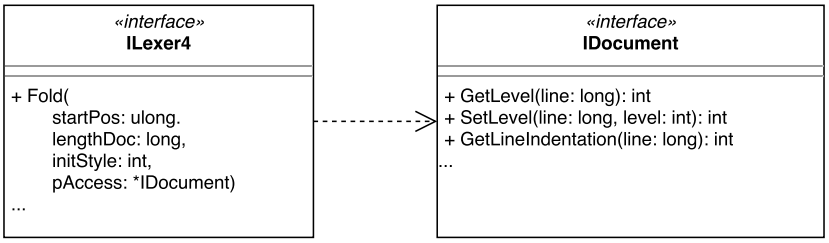


Figure A.3: Overview of the API for code folding in Notepad++.

implementation is expected to call the `setLevel` method on the given `IDocument`. This sets the folding level for a source code line, and tells the editor that lines with the same level can be folded together. Therefore, Notepad++ can only fold whole lines. A collapsed region of code in Notepad++ shows the first line of the region, and this cannot be changed. (Scintilla 2018a)

Eclipse

Eclipse uses a *projection viewer* to project parts of the document to the editor and hide the parts of the document that have been folded. The projection viewer needs to be told which areas are foldable, and it does this through annotations, which are comparable to tags in the Visual Studio model. The API for code folding is shown in Figure A.4.

An Eclipse editor has an associated `ProjectionAnnotationModel`, which is a specification of all the annotated regions. The foldable regions in the code can be changed by modifying the annotation model through the `modifyAnnotations` method, which adds new regions, removed deleted regions, and updates the remaining regions. This ensures annotations (and therefore the state of foldable regions) are kept throughout the editing session. Furthermore, the annotation model has methods to control the expansion state of a foldable region. (Deva 2005)

IntelliJ

The foldable regions of code in IntelliJ are specified by a *folding builder* which returns a descriptor for each foldable range of text. When foldable regions are part of the same group, they collapse and expand together. The relevant API is shown in Figure A.5.

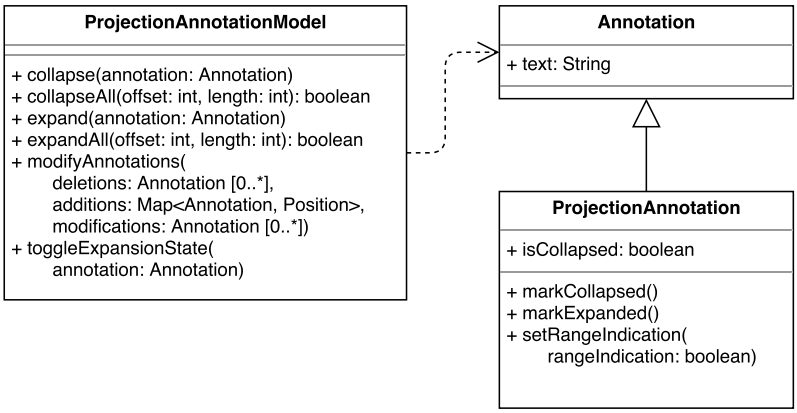


Figure A.4: Overview of the API for code folding in Eclipse.

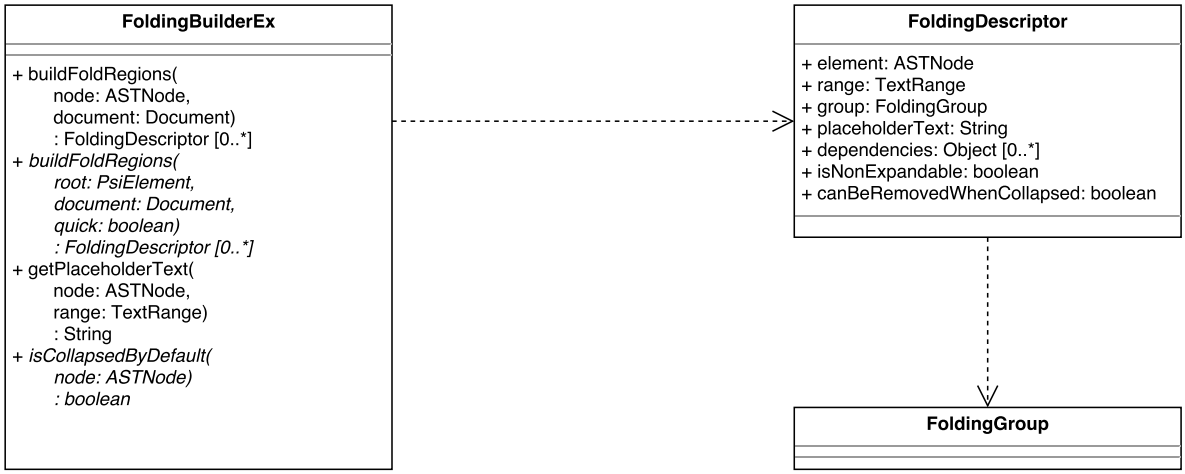


Figure A.5: Overview of the API for code folding in IntelliJ.

To support code folding, the `FoldingBuilderEx.buildFoldRegions` method is the main method to be implemented. Given a Program Structure Index (PSI) element in a document, the implementation of `buildFoldRegions` must return a collection of `FoldingDescriptor` objects, one for each foldable range within the boundaries of the PSI element. The descriptor specifies, among other things, the placeholder text to show when the region is collapsed, and the folding group to which the region belongs. Folding regions that belong to the same group are collapsed and expanded together. (JetBrains 2018q)

Eclipse Che

Eclipse Che internally uses the Orion editor component, which has an annotation model similar to that of Eclipse. Folding annotations can be added and removed from the annotation model, and a given folding annotation can be expanded or collapsed, as shown in Figure A.6.

Upon a change to the current document, a simple implementation would call `getAnnotationModel` to get the annotation model of the editor, and remove any previously added folding annotations. With the annotation model clear of folding annotations, the implementation can analyze the document's text and add a folding annotation to the annotation model for each foldable region in the code. The start and end of a foldable region is specified as an absolute character offset, and this allows the foldable region to start or end in the middle of a line. (Eclipse 2017b)

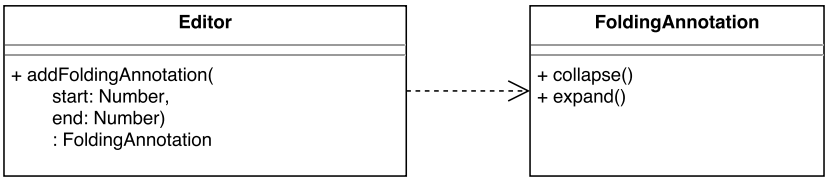


Figure A.6: Overview of the API for code folding in Eclipse Che's Orion editor.

A.1.2 Feature Comparison

Feature support varies between the five editors that have an API for code folding. As shown in Table A.7, Visual Studio and IntelliJ support most features, such as a custom label and default state for the folded region. Except for Notepad++, foldable regions in these editors do not necessarily need to fold whole lines; the regions can start and end at arbitrary characters within a line.

	VS	NP++	Eclipse	IntelliJ	Che
Code folding	●	●	●	●	●
Collapsed region					
Custom label	●	○	○	●	○
Custom tooltip	●	○	○	○	○
Default state	●	○	●	●	○
Character-based	●	○	●	●	●
Folding groups	○	○	○	●	○
Implementation	●	○	○	○	○

Table A.7: Comparison of code folding editor service features across editors with a dedicated API. (●, customizable through API; ○, no API)

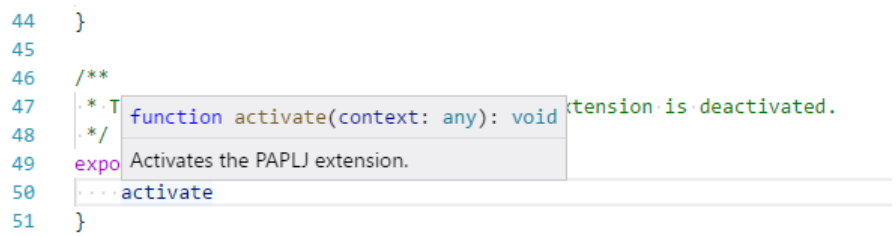


Figure A.8: Hover documentation in VS Code shows a tooltip with the signature and description of the function under the caret.

A.2 Hover Documentation

Hover documentation is an editor service that shows a tooltip with documentation when the user hovers their mouse over a keyword or identifier in their source code. The documentation assists the user in getting a quick understanding of the code, as shown in Figure A.8.

A.2.1 Editor Support

More than half of the editors we looked at support hover documentation. Notepad++, however, can only show documentation for pre-defined API calls whose signatures have been written in an XML file. Sublime, Atom, and Vim do not natively support hover documentation at all.

Visual Studio

In Visual Studio, a *quick info source*, created by its corresponding provider class, provides the documentation to show in a tooltip when the mouse cursor is hovering over a recognized identifier. Usually the tooltip displays plain text by converting the given quick info content object to its string representation, but it is possible to return a `UIElement` instead that contains styled text. The quick info source returns a span, which indicates the region of source code being described. The API is shown in Figure A.9. (Microsoft 2016d)

Eclipse

The plugin has to implement the `ITextHover` and `ITextHoverExtension2` interfaces. (Breslau 2009b) By using the appropriate implementation of the `HoverControl`, rich text hover can be achieved. The relevant API is shown in Figure A.10. (Breslau 2009a)

IntelliJ

A plugin in IntelliJ can provide a documentation tooltip for a given PSI element. When the PSI element is a reference, documentation is requested for its definition instead. Similarly, the user can request a more extensive documentation popup which is provided by the same class. The documentation can be styled. The API is shown in Figure A.11. (JetBrains 2018e)

VS Code

VS Code requires the implementation of a `HoverProvider`, which, when given the document and a location, returns the documentation as text with layout. The API is shown in Figure A.12. (Microsoft 2018c) Alternatively, the hover provider can be implemented as part of a Language Server Protocol (LSP) language server. (Microsoft 2018e)

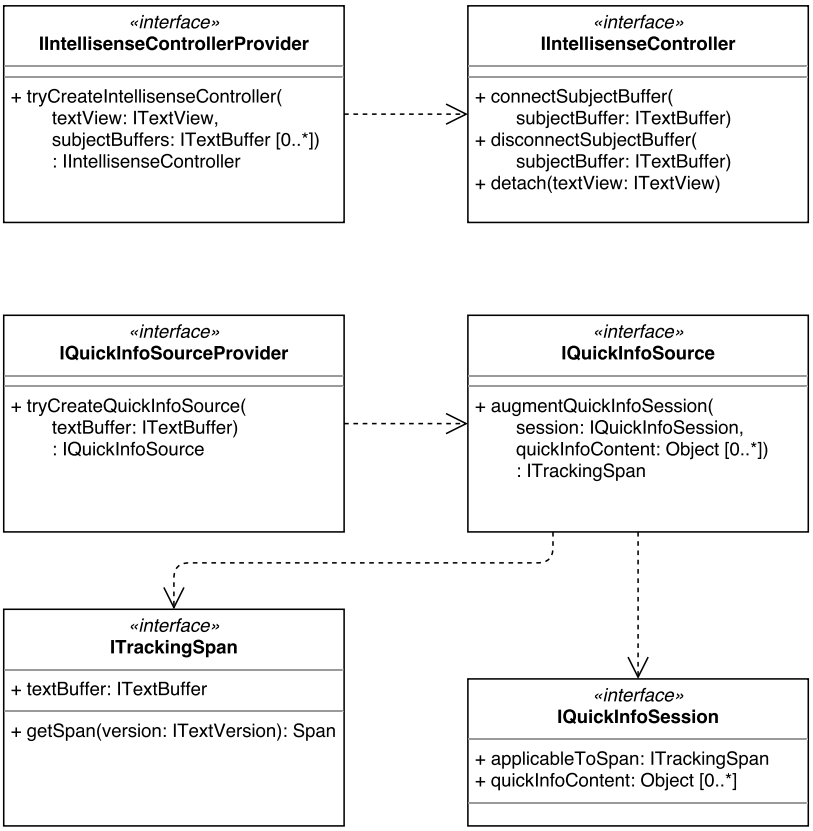


Figure A.9: Overview of the API for hover documentation in Visual Studio.

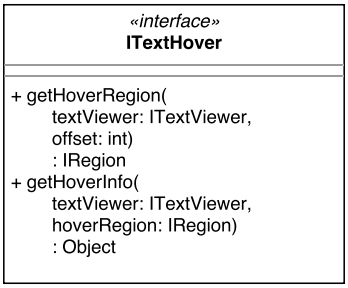


Figure A.10: Overview of the API for hover documentation in Eclipse.

Cloud9

A custom language in Cloud9 can show signature help for the symbol at the current caret location. The tooltip itself can show a function signature with (possibly formatted) documentation, and for each function parameter additional (formatted) documentation. See Figure A.13. (Cloud9 2018d)

Eclipse Che

Eclipse Che can show tooltips (hover documentation) for the symbol under the caret through LSP. When the language server gets a *hover* request with a caret location, it should respond with a styled text and a range of text to highlight in the document to indicate the symbol. This is shown in Figure A.14. (Microsoft 2018a)

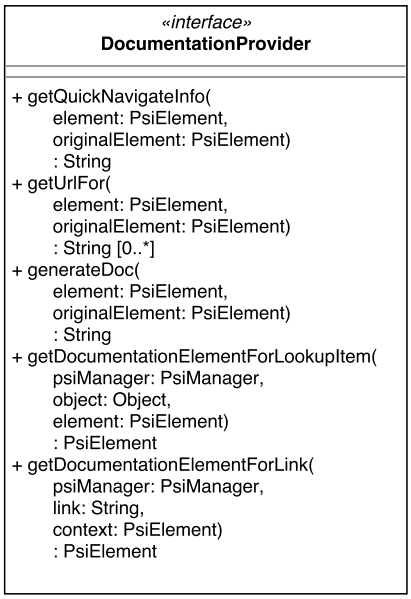


Figure A.11: Overview of the API for hover documentation in IntelliJ.

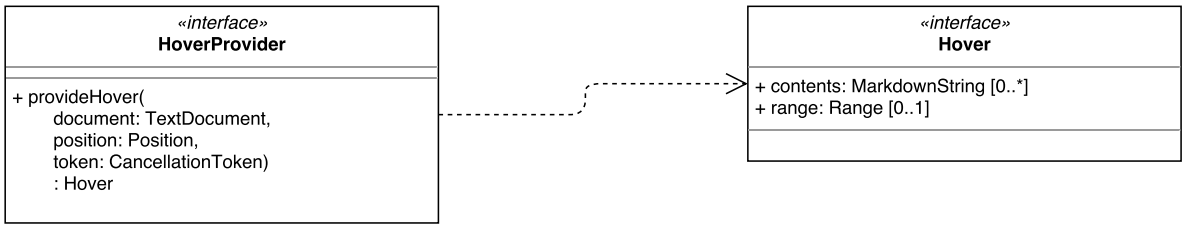


Figure A.12: Overview of the API for hover documentation in VS Code.

A.2.2 Feature Comparison

Hover documentation is a simple editor service with an equally simple API across editors. The editor service is given the current document and caret location, and should return the documentation to display and the document range to which it pertains. Most editors also support styled documentation, although the exact format differs from editor to editor. The features across editors are shown in Table A.15.

	VS	Eclipse	IntelliJ	VS Code	Cloud9	Che
Hover documentation						
Plain documentation	●	●	●	●	●	●
Styled documentation	●	●	●	●	●	●
Markdown documentation	○	○	○	●	○	●
HTML documentation	○	○	○	○	●	○
Symbol region	●	●	●	●	○	●

Table A.15: Comparison of hover documentation editor service features across editors with programmable support.
(●, dedicated API; ○, not supported)

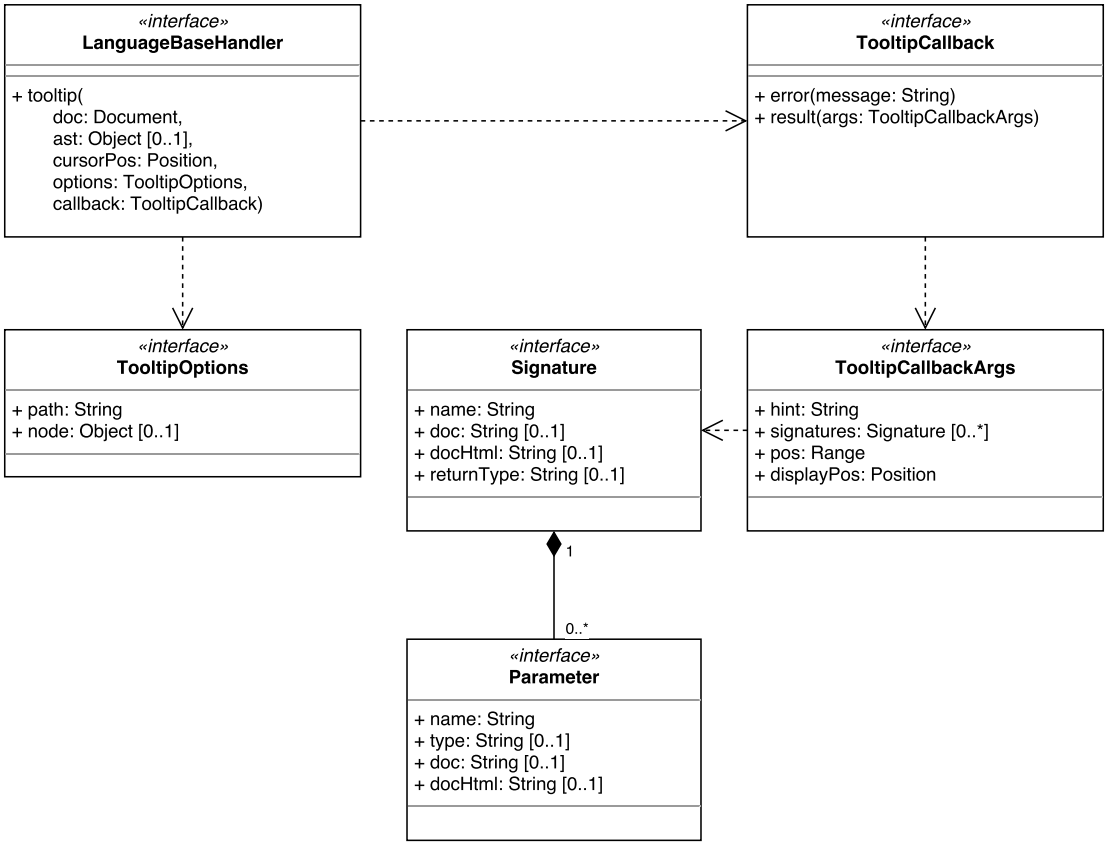


Figure A.13: Overview of the API for hover documentation and signature help in Cloud9.

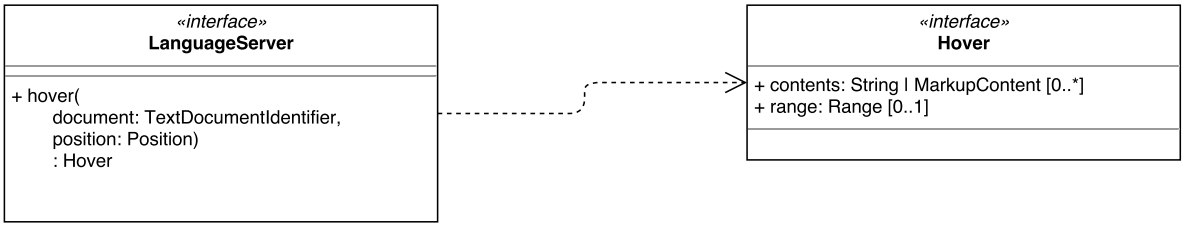


Figure A.14: Overview of the API for hover documentation in LSP.

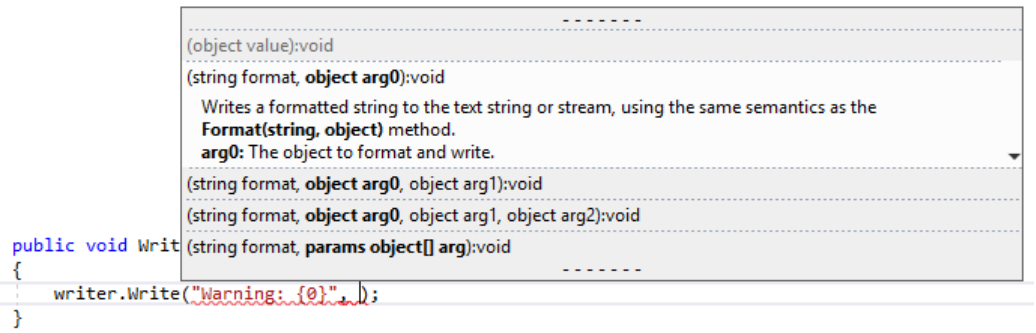


Figure A.16: Signature help in Visual Studio, highlighting the second parameter of the function.

A.3 Signature Help

Signature help shows the parameter names and types, return type, and documentation of the function the user is currently editing. When there is more than one possible signature for the given function, for example in languages that support method overloading, the editor shows all of them. The user selects the overload they want, which is then expanded to show additional documentation. While the user is editing the function's parameters, the editor highlights the current parameter in the signature help and shows the parameter's documentation, as shown in Figure A.16.

A.3.1 Editor Support

Vim and Sublime do not support signature info natively, and Atom is looking to add support for LSP signature info in the future as part of its Atom IDE UI effort. (Wang 2017)

Notepad++ only supports signature info for functions pre-defined in an Extensible Markup Language (XML) file. For each function, Notepad++ accepts one or more overloads, which specify the return type and a description for the overload. An overload has a list of parameters, specifying the name and type of the parameter. (Notepad++ 2014)

Signature help can be implemented in the order editors, where only Eclipse does not have a dedicated API for signature info. Signature help in Eclipse can be implemented manually, which is how signature info support is provided for the Java language.

Visual Studio

Visual Studio employs its *quick info* editor service for function signatures as well, where it displays the function's name and signature along with specific information about the parameter currently under the caret. A *signature help source* returns the applicable signatures, with the styled string to display and any documentation, and for each signature the parameters. Each parameter has its associated name and documentation. If more than one signature applies to a given function (for example when the function is overloaded), the service is asked to determine the most likely match out of all signatures, such that the most likely match is the signature that is expanded and displayed to the user. The API is shown in Figure A.17. (Microsoft 2016e)

IntelliJ

In IntelliJ an implementation of a `ParameterInfoHandler` is given the current document and caret offset, and returns or updates a list of function overloads and the currently highlighted parameter, as shown in Figure A.18. The handler has control over the parameter text and

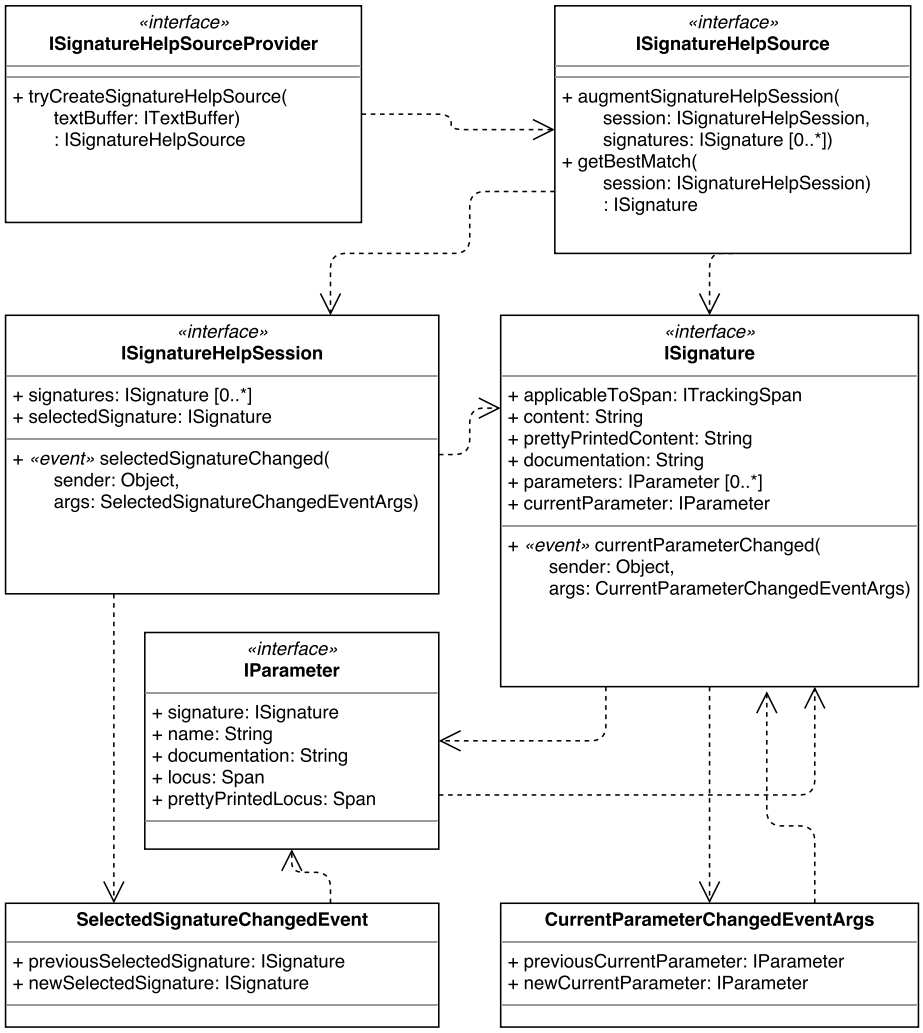


Figure A.17: Overview of the API for signature help in Visual Studio.

some style attributes, including the background color, and whether it is grayed or struck out. (JetBrains 2018a)

VS Code, Eclipse Che and LSP

The API for signature help in VS Code and LSP (and therefore Eclipse Che) are very similar. Both services get the document and caret location, and return one or multiple function signatures, with a list of parameters for each signature. The service can indicate the active signature and parameter, and each signature and parameter has an associated label and optional documentation. Only the VS Code API also accepts a cancellation token, as shown in Figure A.19. (Microsoft 2018e; Microsoft 2018c) The LSP API, supported by both VS Code and Eclipse Che, is shown in Figure A.20. (Microsoft 2018a)

Cloud9

The API for signature help in Cloud9 has been shown previously in the context of hover documentation in Figure A.13. It displays the function signatures, each of which can have parameters. Both can have associated documentation, which may be styled using HyperText Markup Language (HTML). (Cloud9 2018d)

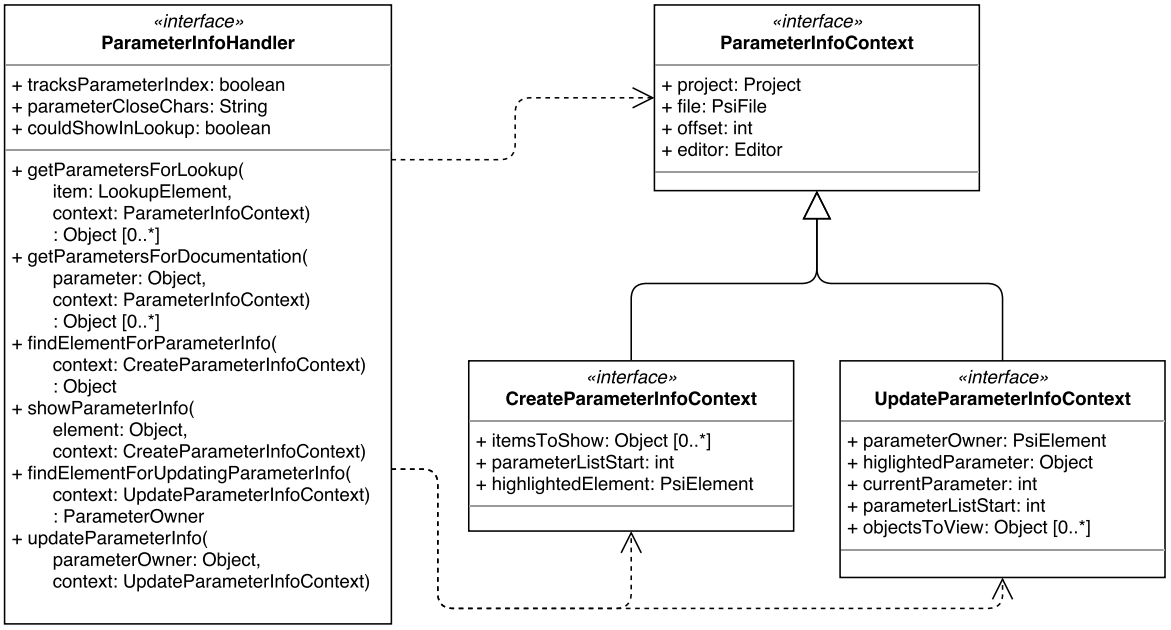


Figure A.18: Overview of the API for signature help in IntelliJ.

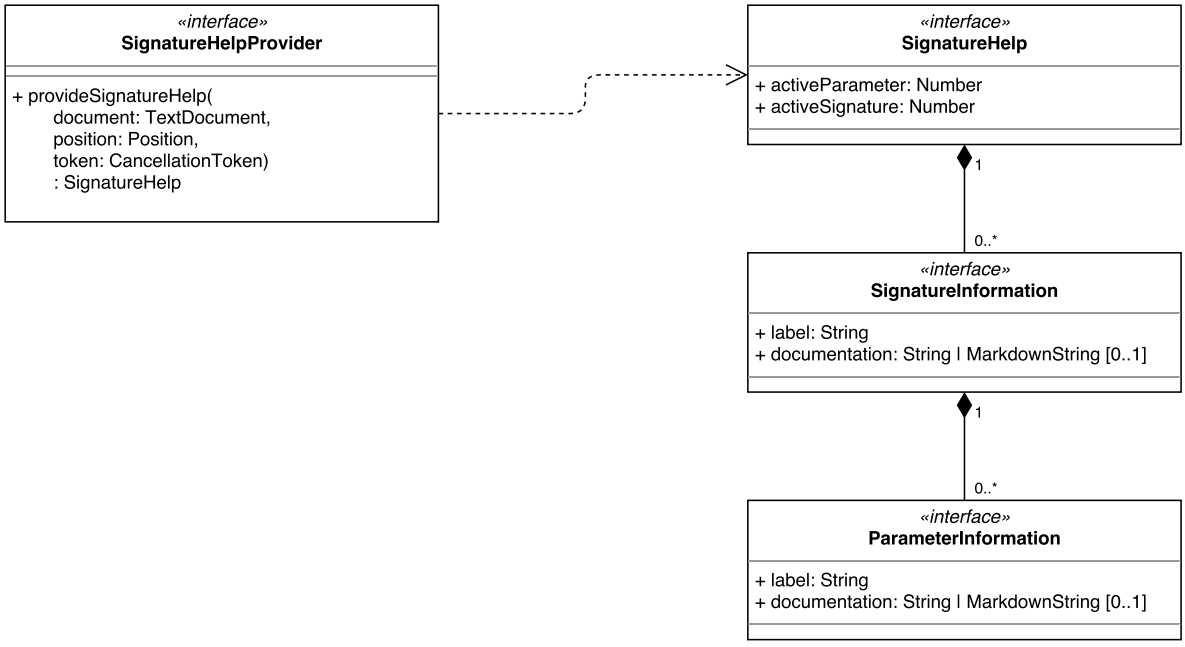


Figure A.19: Overview of the API for signature help in VS Code.

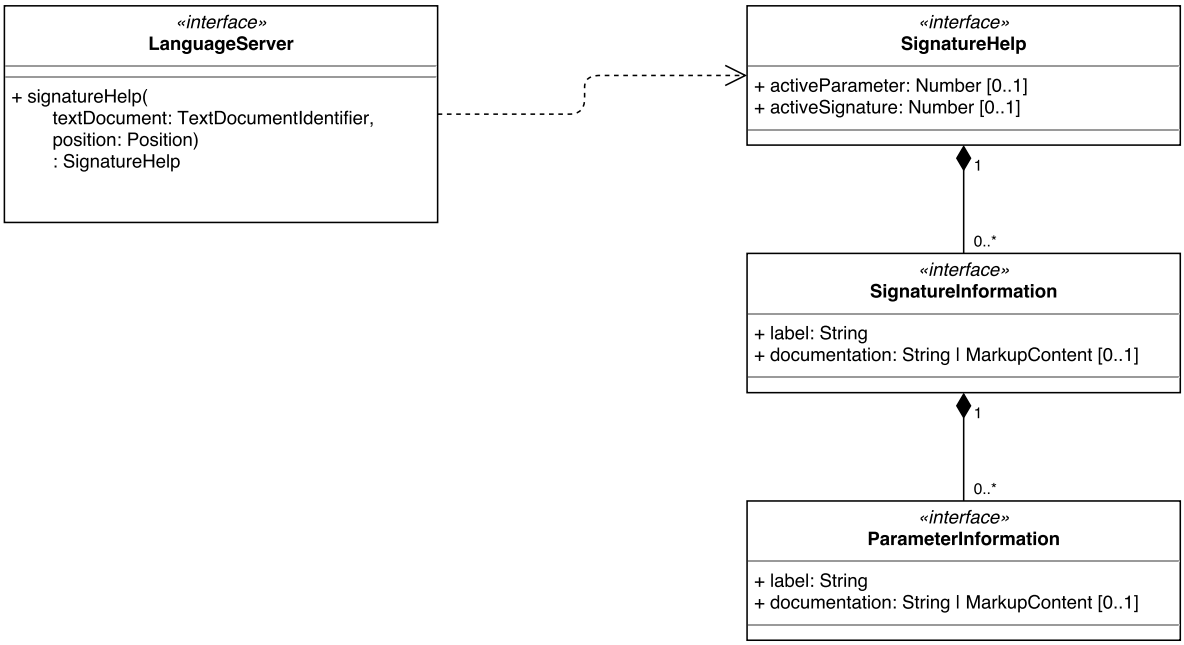


Figure A.20: Overview of the API for signature help in LSP.

A.3.2 Feature Comparison

Most code editors have built-in support for showing signature information, and Table A.21 shows the features each editor supports. Specifying the return type of the function or the type of a parameter is not explicitly supported in VS Code or LSP (and therefore not in Eclipse Che), but most languages circumvent this by concatenating the name and the type together to form the label displayed to the user. Rich documentation — that is, text with layout such as monospaced fonts and bold or italic styles — is only supported by Cloud9, VS Code and LSP.

	VS	IntelliJ	VS Code	Cloud9	Che
Signatures					
Label	●	●	●	●	●
Return type	●	○	○	●	○
Documentation	●	●	●	●	●
Styled documentation	○	○	●	●	●
Multiple overloads	●	●	●	●	●
Parameters					
Label	●	●	●	●	●
Type	●	○	○	●	○
Documentation	●	●	●	●	●
Styled documentation	○	○	●	●	●

Table A.21: Comparison of signature help editor service features across editors with programmable support.
(●, dedicated API; ○, not supported)

```
1   run let
2   Fib fib = new Fib()
3   Num n   = 5
4   in
5     {
6       fib.initialize( n );
7         fib.value =0;     // default value
8         fib.next.value = 1;
9       fib.calculate( n );
10      fib.getValue( n )
11      ; }
```

```
1   run
2    let
3      Fib fib = new Fib()
4      Num n = 5
5     in {
6       fib.initialize(n);
7       fib.value = 0;     // default value
8       fib.next.value = 1;
9       fib.calculate(n);
10      fib.getValue(n);
11  }
```

Figure A.22: Automatic formatting in Visual Studio changes the whitespace and line breaks in the left-hand code into those displayed at the right, for a consistent source code layout and improved readability.

A.4 Automatic Formatting

Applying a consistent formatting to the source code is essential in keeping the code base consistent across developers. Automatic formatting provides three editor services that assist with this goal: document formatting, range formatting, and live formatting. The first two fix the formatting of (part of) the current document, of which an example is shown in Figure A.22. Live formatting applies the formatting rules while the user types, such that, for example, typing a newline in the middle of an expression will correctly indent the next line, and typing a closing curly brace will format the code block that has just been closed and put the closing curly brace itself on the correct line.

A.4.1 Editor Support

A dedicated API for automatic formatting is only present in IntelliJ, Cloud9, VS Code and Eclipse Che. While there is no such API in Eclipse and Visual Studio, a plugin can add automatic formatting support by listening to IDE events, such as when the user invokes a *Format* command or types a special character, and then modify the source code to reflect the correct formatting. Sublime, Atom, Notepad++, and Vim do not support automatic formatting at all.

IntelliJ

In IntelliJ, the plugin has to implement a *formatter* that builds a tree of blocks, where each block covers some of the PSI elements of the current document. A custom *spacing builder* will assign the appropriate spacing rules to the blocks, such as indentation, blank lines, and whitespace. The API is shown in Figure A.23. (JetBrains 2018g)

This combination of a formatter and spacing builder is also used to provide live code formatting while the user writes. IntelliJ does not support using an external formatter. (JetBrains 2018d)

VS Code

A plugin for VS Code needs to implement the `DocumentFormattingEditProvider`, `DocumentRangeFormattingEditProvider` and `OnTypeFormattingEditProvider` to support automatic and live formatting. Each provider returns the minimal collection of text edits that is required to apply the new formatting. Returning the minimal collection is important, as it influences the

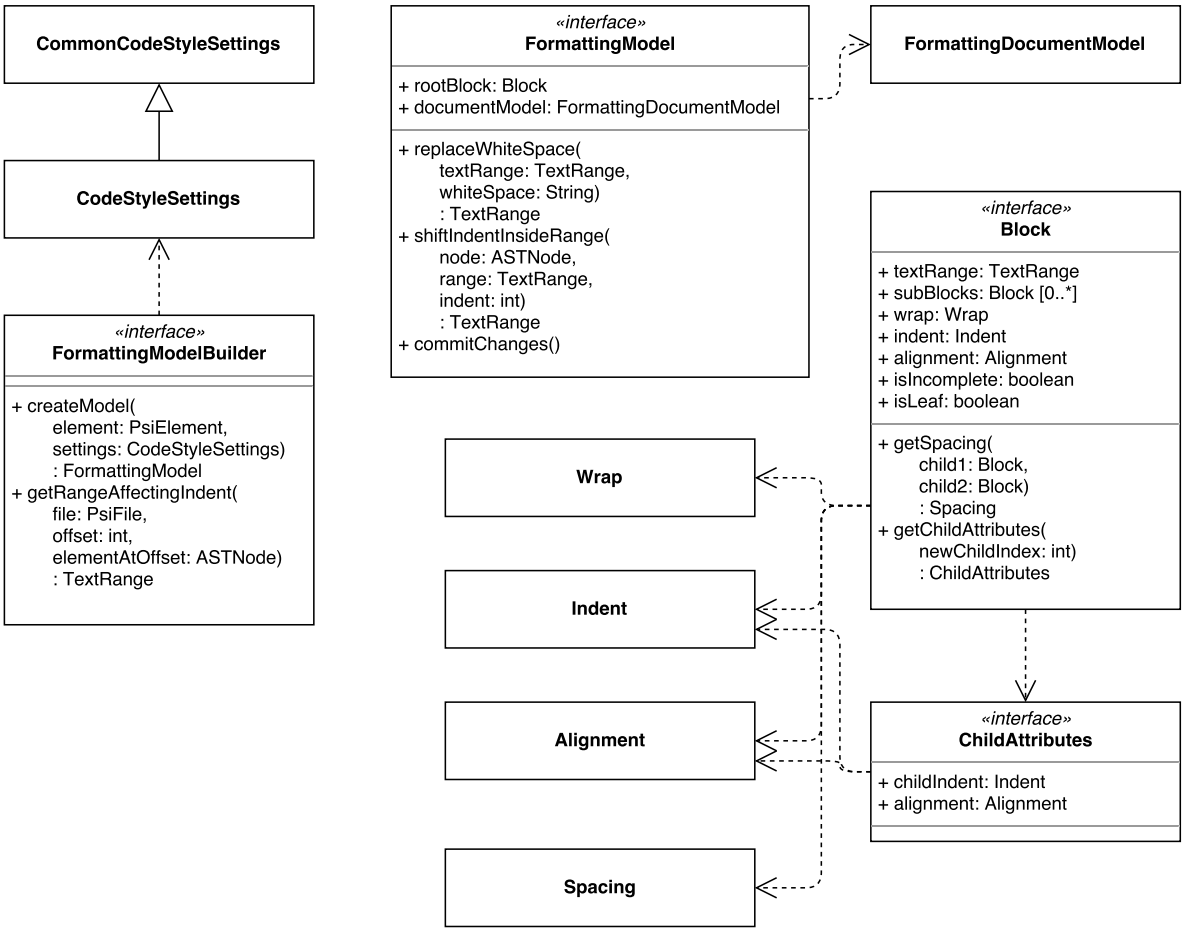


Figure A.23: Overview of the API for automatic formatting in IntelliJ.

caret location and markers if done incorrectly. The API is shown in Figure A.24. (Microsoft 2018e)

Cloud9

Cloud9 supports automatic formatting for the whole document through a formatting handler, where the handler is called for a specific document to format, and should return the formatted document. Formatting part of a document and live formatting are not supported. The API is shown in Figure A.25. (Cloud9 2018d)

Eclipse Che

Eclipse Che supports formatting a whole document, range of a document, or live formatting through LSP. As part of a formatting request, the server is given the document, the range (if formatting a range) or the caret location (if live formatting), and a list of formatting options, such as whether to insert spaces and how many for each tab. The server responds with a list of text edits that the client has to perform on the document to format it properly. The API is very similar to that of VS Code, and shown in Figure A.26. (Microsoft 2018a)

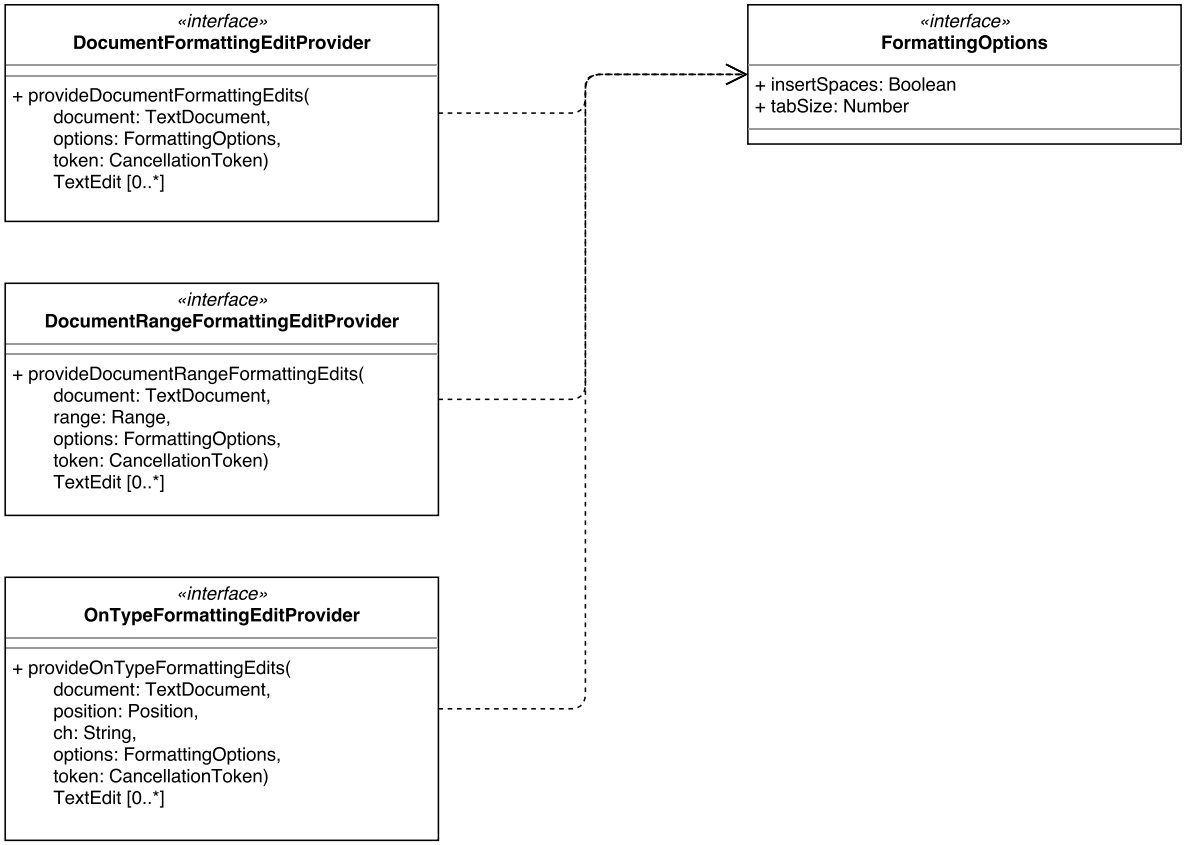


Figure A.24: Overview of the API for automatic formatting in VS Code.

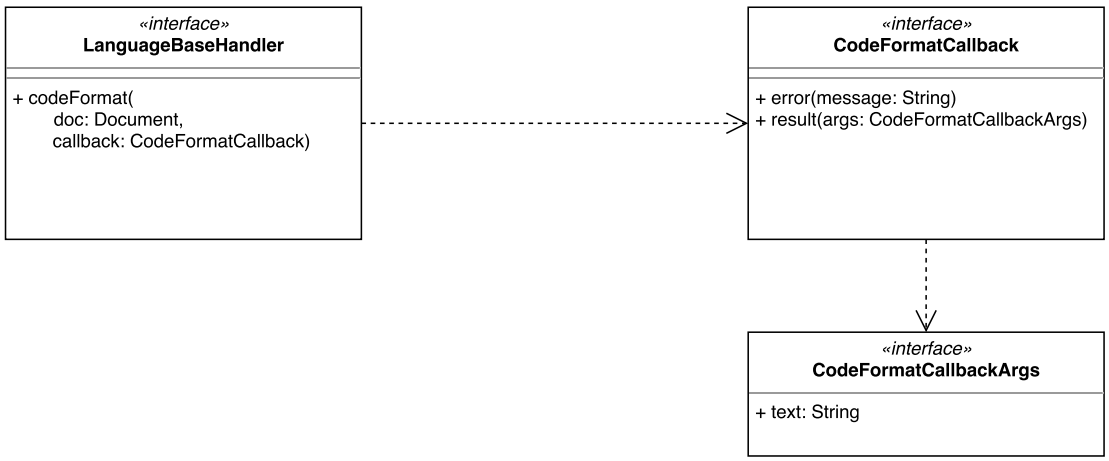


Figure A.25: Overview of the API for automatic formatting in Cloud9.

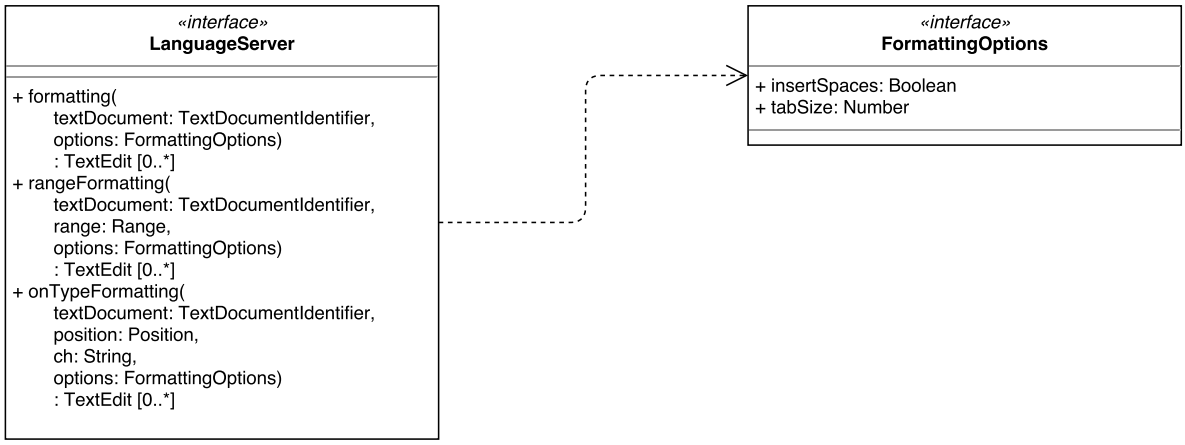


Figure A.26: Overview of the API for automatic formatting in LSP.

A.4.2 Feature Comparison

The APIs for automatic formatting in VS Code and Eclipse Che are nearly the same, and support document formatting, range formatting, and live formatting. Cloud9 has a handler only for document formatting, and IntelliJ has a complex system that works on the document's abstract syntax tree (AST), but which does not support an external formatter. None of the editors expose an API that allows the user to set the formatting preferences. Their features are shown in Table A.27.

	IntelliJ	VS Code	Cloud9	Che
Formatting				
Document formatting	●	●	●	●
Range formatting	●	●	○	●
Live formatting	●	●	○	●
External formatter	○	●	●	●

Table A.27: Comparison of automatic formatting editor service features across editors with programmable support.
(●, dedicated API; ○, not supported)

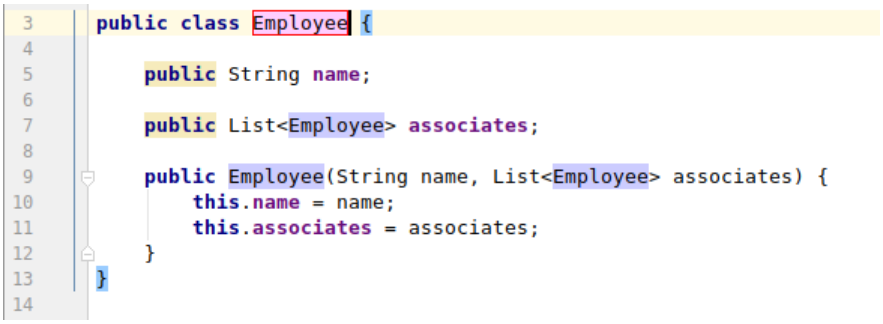


Figure A.28: Rename refactoring in IntelliJ shows a live preview of the code resulting from changing the name of the class.

A.5 Rename Refactoring

Refactoring is the process of restructuring the source code without changing its semantics, and rename refactoring is a special case where changing a declaration's name also changes all references across the code files in a project. Rename refactoring is smarter than a simple find-and-replace, as the refactoring would not replace names that are not actually referring to the declaration being renamed.

An editor can either show a dialog where the user can input the new name, or let the user type the name directly in the source code. Some editors, such as IntelliJ, can show a live preview of the resulting code, as shown in Figure A.28.

A.5.1 Editor Support

Visual Studio does not have a dedicated API for rename refactoring, and Notepad++, Sublime, Vim, and Atom do not support rename refactoring at all. The other editors support rename refactoring to various degrees.

A.5.2 Eclipse

Rename refactoring is one of the refactorings Eclipse supports out-of-the-box. (Eclipse 2018e) A plugin would have to implement a *rename participant*, which can check whether it can apply the renaming, and return the changes that must be applied to the documents as a result of renaming a definition or declaration. The API is shown in Figure A.29.

IntelliJ

IntelliJ supports rename refactoring out of the box. IntelliJ's reference resolution system is used to determine which PSI elements need to be renamed. A custom language plugin can provide a name validator in the case of renaming. By providing a custom `RenameHandler` implementation, IntelliJ's default implementations can be subverted and custom logic can be implemented, as shown in Figure A.30. (JetBrains 2018k)

VS Code

A rename refactoring in VS Code gets a document, caret location, new name, and cancellation token, and returns a workspace edit, which is a collection of text edit entries that change some specific range of text in a document, usually to replace the old name by the new. The API is shown in Figure A.31. (Microsoft 2018c)

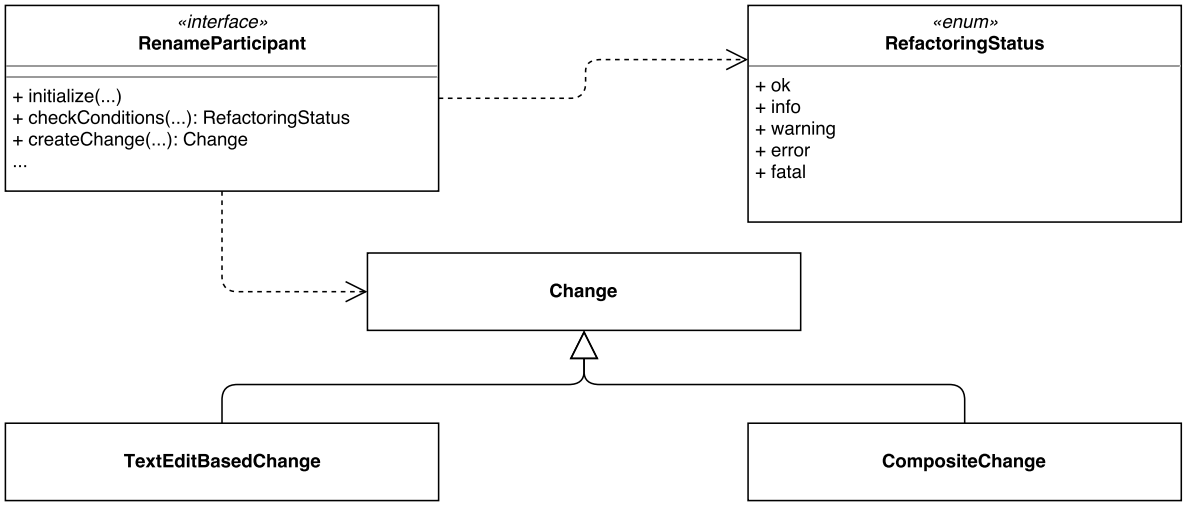


Figure A.29: Overview of the API for rename refactoring in Eclipse.

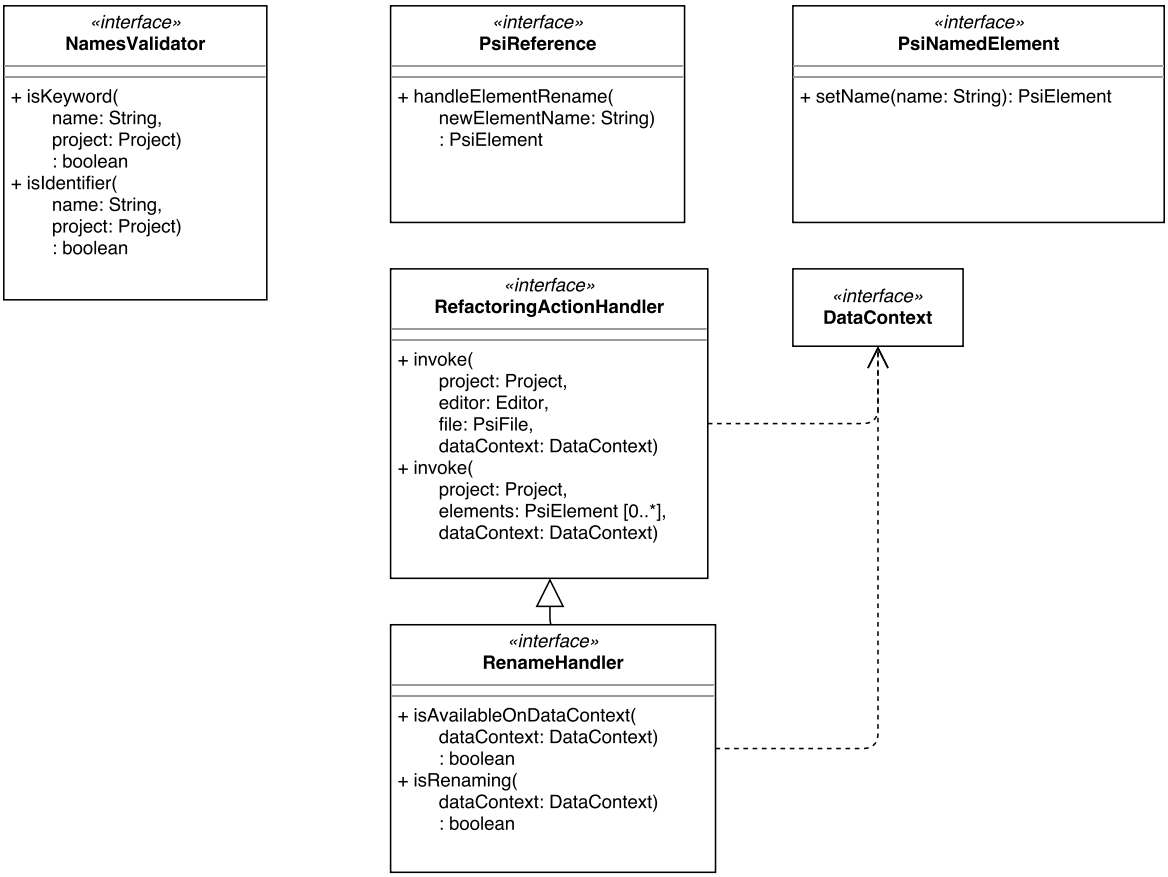


Figure A.30: Overview of the API for rename refactoring in IntelliJ.

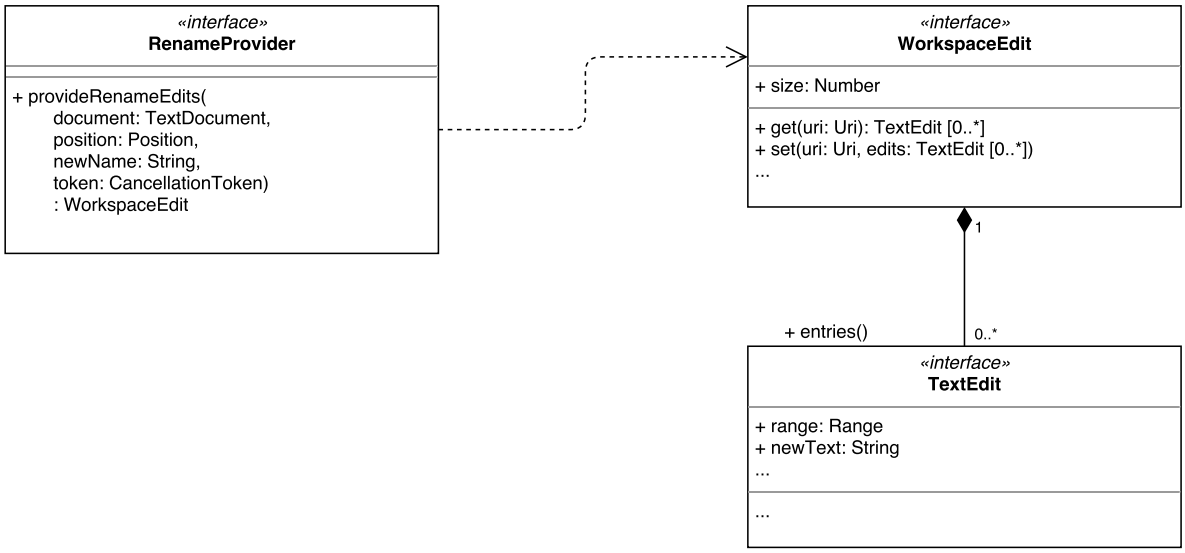


Figure A.31: Overview of the API for rename refactoring in VS Code.

Cloud9

The refactorings handler of Cloud9 returns a list of refactorings available at the current caret location in the document, but rename refactoring is currently the only supported refactoring.

When a symbol is renamed, the rename handler lists the span of the symbol being renamed, and the spans of all other symbols across the project where the same renaming must be applied. Before committing to a name, its validity can be checked, for example that the new name is not a reserved keyword or uses illegal characters. The API is shown in Figure A.32. (Cloud9 2018d)

Eclipse Che

Eclipse Che supports rename refactoring through LSP. The request is a document location and the new name for the symbol, and the response is a list of edits to make across documents in the workspace, or an error when the new name is not allowed. The relevant API is shown in Figure A.33. (Microsoft 2018a)

A.5.3 Feature Comparison

Rename refactoring is relatively simple, and therefore supported across editors in similar ways. The IntelliJ API is the least flexible, as it works on the level of the code AST, whereas the other editors allow arbitrary text edits for a refactoring. See the features in Table A.34.

	Eclipse	IntelliJ	VS Code	Cloud9	Che
Rename refactoring					
Text edits across project	●	●	●	●	●
Custom name validation	●	●	●	●	●

Table A.34: Comparison of rename refactoring editor service features across editors with programmable support.
(●, dedicated API; ○, not supported)

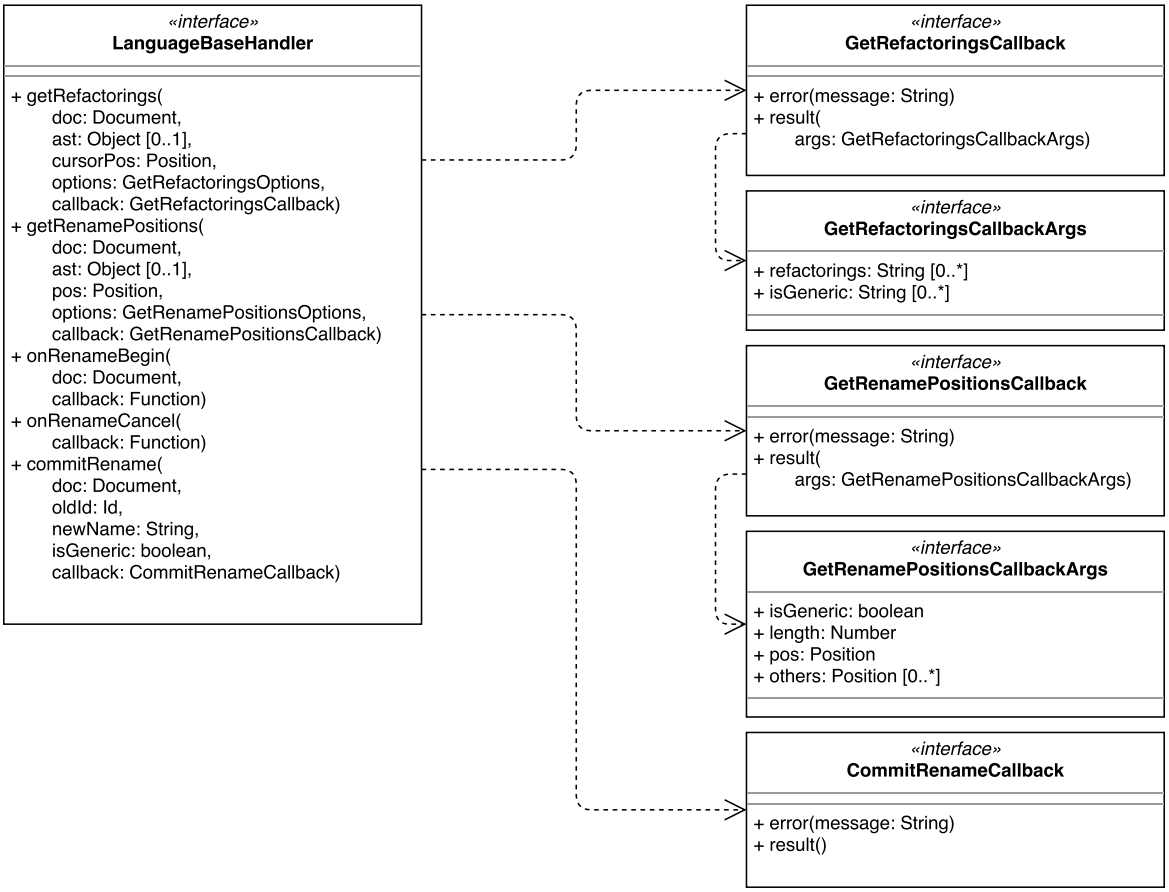


Figure A.32: Overview of the API for rename refactoring in Cloud9.

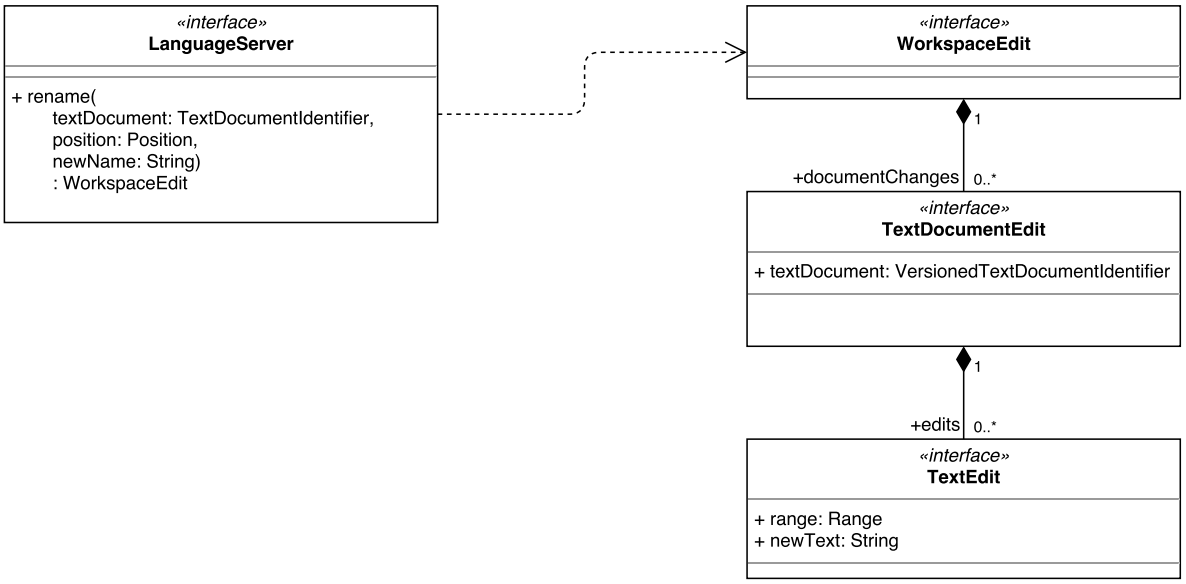


Figure A.33: Overview of the API for rename refactoring in LSP.

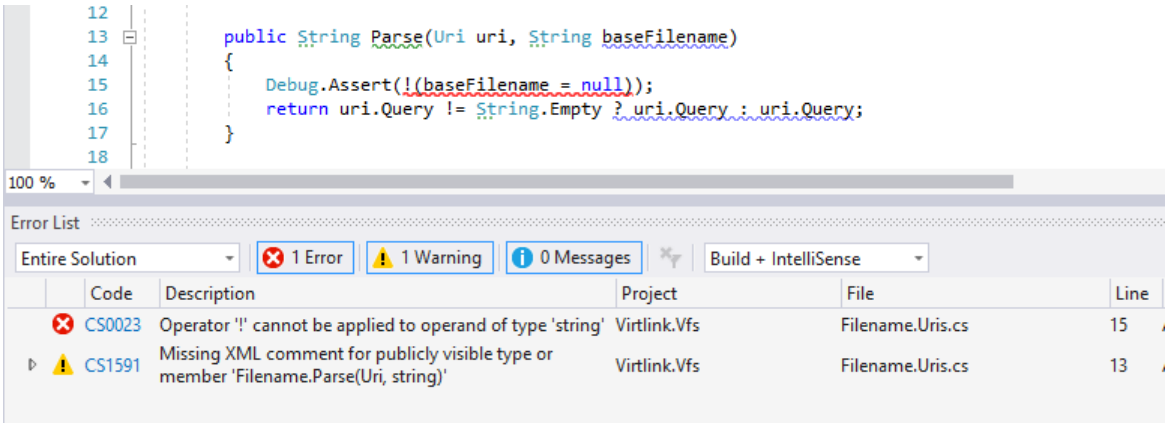


Figure A.35: Diagnostic messages in Visual Studio show errors, warnings, and suggestions as red, blue, and green squiggles respectively, and hints as three dots under a word. The error list shows a summary of the most important errors and warnings in the project.

A.6 Diagnostic Messages

Upon building the program, the compiler may generate diagnostic messages such as warnings and errors. Most of these, especially syntactic and semantic messages, will be related to some part of the source code. Editors can gather all these messages in one list, and show the locations of errors and warnings with squiggles in the source code, as shown in Figure A.35.

A.6.1 Editor Support

There are two main ways in which editors support custom diagnostic messages. The simplest, used by VS Code, Vim, and Sublime, is when the editor invokes a command-line tool, and then parses the output to extract the diagnostic messages. Other editors, however, support custom analysis or annotation logic. Only Notepad++ and Atom do not support custom diagnostic messages at all.

VS Code uses *problem matchers* to read warnings and errors from the output of tasks. A problem matcher is a regular expression that deconstructs the output into their file, line, column, severity (warning or error), and message text, and optionally the end line, end column, and error code. These patterns can be nested to capture across multiple lines. (Microsoft 2018d)

Sublime has a similar system where a regular expressions is used to capture the filename, line number, column number, and error message from the output of the build system. However, an error or warning can only apply to whole lines. (Sublime Text 2018b)

In Vim their *compiler plugins* can define the command and arguments to invoke, and a special expression defining the error format. (Vim 2011)

Visual Studio

Diagnostic messages in Visual Studio can come from various sources. The build script can return messages, and the parser and extension services can add messages. The latter do this through the `ErrorListProvider`, to which *error tasks* can be added that each represent the diagnostic message, its file and location, and category. The API is shown in Figure A.36. (Kinable 2010)

The tools that are invoked by tasks in an MSBuild build script can return messages of their own. If the tool is called as part of a custom task implementation, the task can handle logging the messages. Alternatively, if the error and warning messages of the tool conform

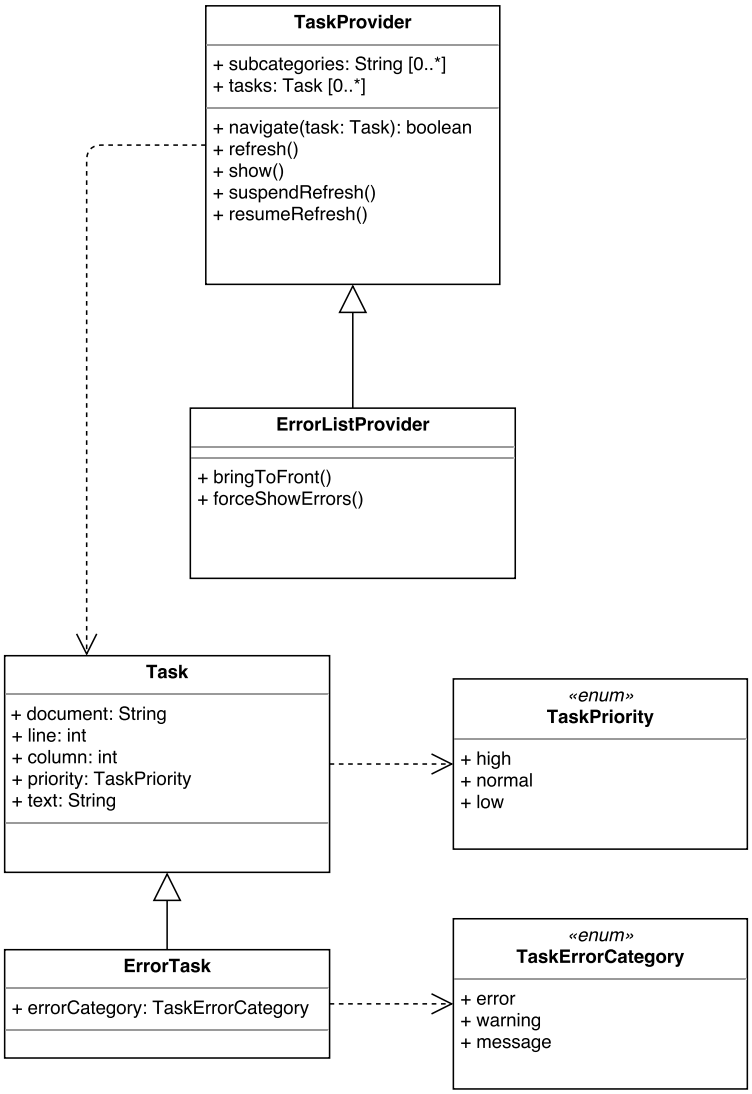


Figure A.36: Overview of the API for diagnostic messages in Visual Studio.

to a specific format, Visual Studio can parse and display the messages itself. A task message can either be an error or a warning, and can specify the file and location within the file to which the message pertains. (Microsoft 2016a)

An example of such a message would be: `C:\myfile.pj(1,2): error: syntax error: expected 'program', got 'import'`, which indicates an error on line 1 column 2 of the `myfile.pj` file.

Eclipse

Eclipse builders and other extensions can report errors, warnings and other diagnostic messages by marking parts of the source document, the document itself, or the whole project, with a marker implementing `IMarker`. This will highlight the specified words and create a note in the *problems view*. A marker carries information about its severity, priority, message, line number, and location. The API is shown in Figure A.37. (Glozic 2001; Eclipse 2018b)

IntelliJ

IntelliJ has several ways to communicate diagnostic messages to the user: when parsing, the lexer can return a special token to indicate an unexpected character or string, or the parser

Figure A.37: Overview of the API for markers in Eclipse.

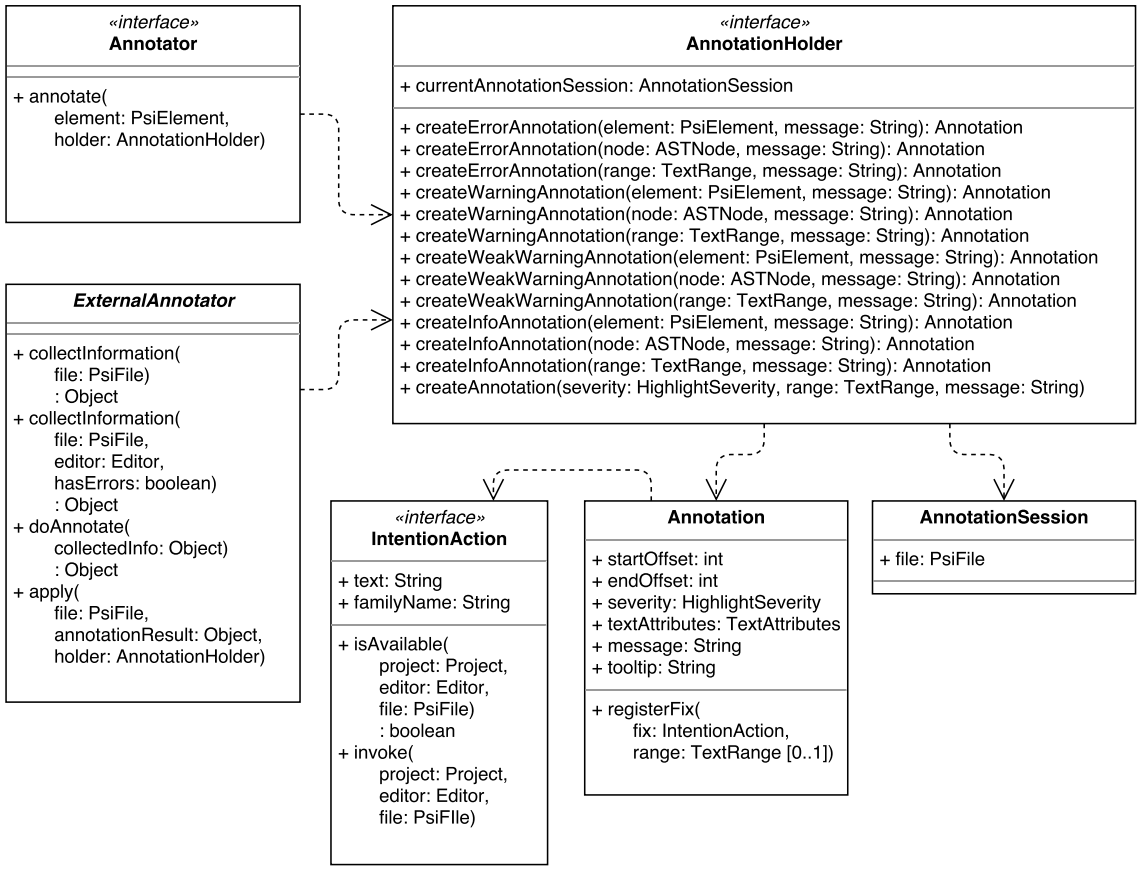


Figure A.38: Overview of the API for annotations in IntelliJ.

can mark part of the AST with a syntax error. (JetBrains 2018o)

Additionally, an *annotator* can be used to annotate ranges of source code with error, warning, weak warning, and information messages. If the annotator needs to invoke an external tool (and therefore takes more time), it can be implemented as an *external annotator*. The API is shown in Figure A.38. (JetBrains 2018b)

Cloud9

Cloud9 can show diagnostic messages after analyzing the current document. A diagnostic message is either an error, warning, or informational message with an associated location in the document. The editor makes a distinction between minimal analysis and full analysis, with the idea that minimal analysis should be quick, for example for code completion or tooltips. The API is shown in Figure A.39. (Cloud9 2018d)

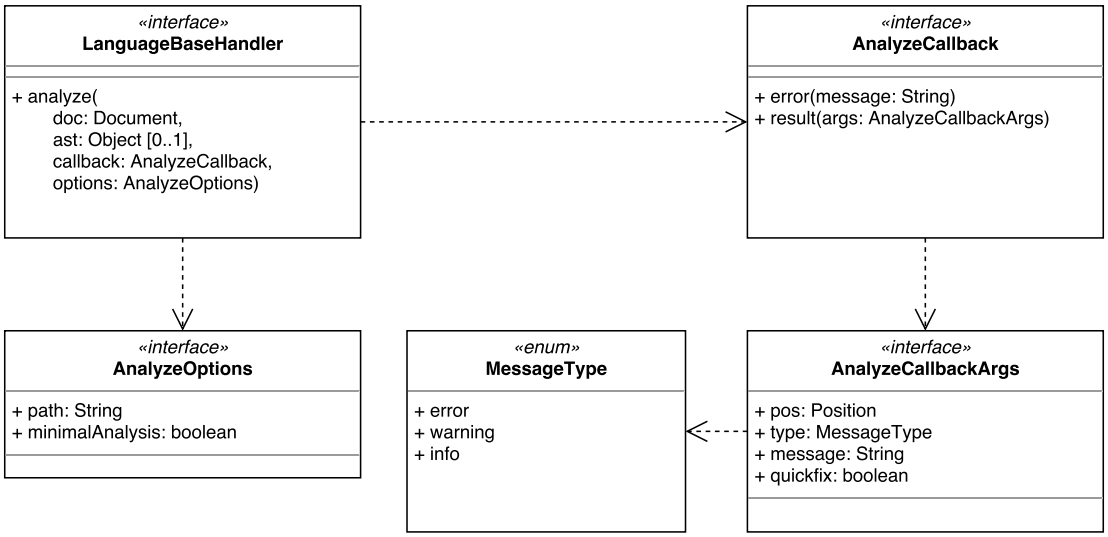


Figure A.39: Overview of the API for diagnostic messages in Cloud9.

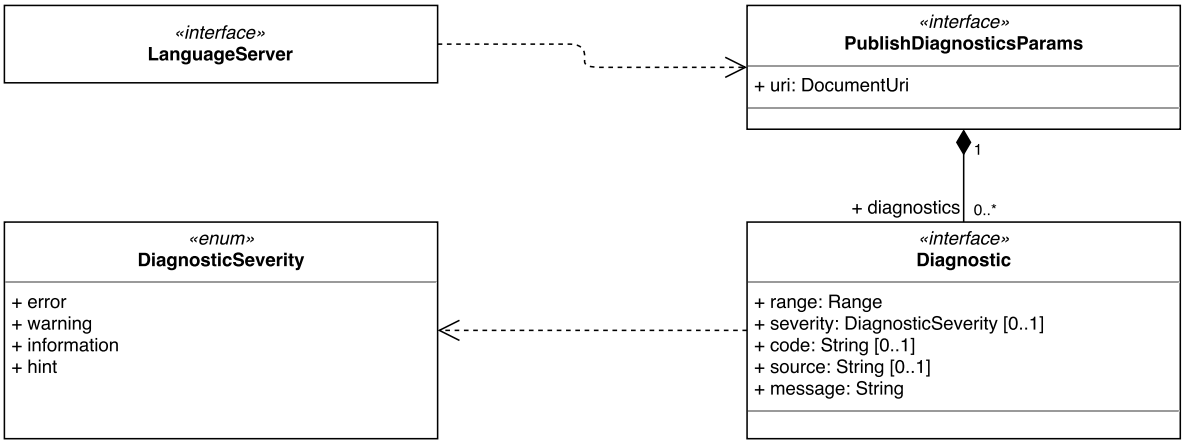


Figure A.40: Overview of the API for diagnostic messages in LSP.

Language Server Protocol

VS Code and Eclipse Che both support LSP, where the language server pushes new diagnostic messages to the editor client whenever they are available. The new messages replace any previous messages for that document.

Each diagnostic messages describes a range of a document, and assigns a message and optional error code to it. A given message can indicate an error, a warning, information, or a hint. The relevant API is shown in Figure A.40.

A.6.2 Feature Comparison

Almost all editors can show both errors and warnings on a line, and most editors provide even more fine-grained control over the placement of diagnostic messages. Editors support either custom logic that returns diagnostic messages, or can parse these messages when they follow a pre-specified format. The full feature comparison is shown in Table A.41.

	VS	Sublime	Vim	Eclipse	IntelliJ	VS Code	Cloud9	Che
Sources								
Parsed messages	●	●	●	○	○	●	○	○
Custom parsed messages	○	●	●	○	○	●	○	○
Custom logic	●	○	○	●	●	●	●	●
Severity								
Error	●	●	●	●	●	●	●	●
Warning	●	○	●	●	●	●	●	●
Weak Warning	○	○	○	○	●	○	○	○
Info	○	○	○	●	●	●	●	●
Hint	○	○	○	○	○	●	○	●
Scope								
Project	○	○	○	●	○	○	○	○
Document	○	○	○	●	○	○	○	○
Line	●	●	●	●	●	●	●	●
Location	●	○	●	●	●	●	●	●
Region	○	○	○	●	●	●	●	●

Table A.41: Comparison of diagnostic messages editor service features across editors. (●, supported; ○, not supported)

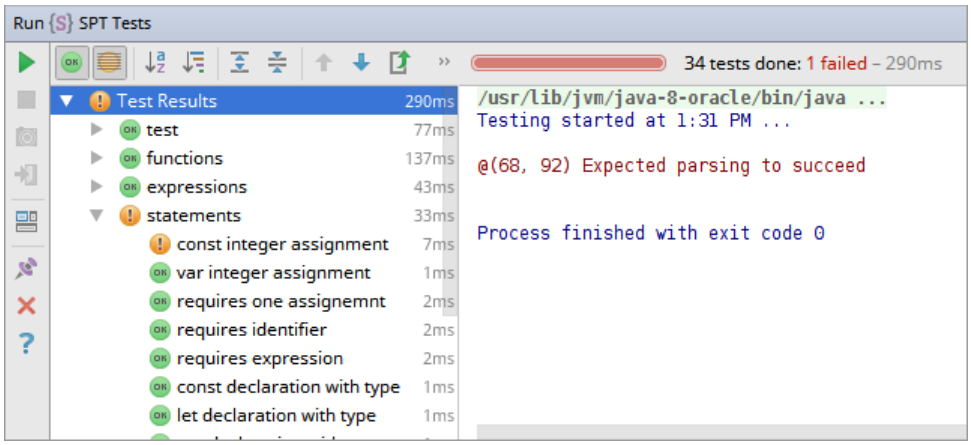


Figure A.42: The test runner in IntelliJ shows a tree of test suites and test cases on the left-hand side, and the details about one or more tests on the right-hand side. The icon of each test in the tree shows its result: one test failed.

A.7 Testing

The ability to write and run tests is an integral part of software development. Tests in general purpose languages are usually written as normal code adorned with some third-party test framework's special constructs and annotations (such as used by JUnit for Java), while other languages have a tightly integrated test framework (such as in Rust). A domain-specific language (DSL) that lacks the expressivity required to support arbitrary tests may provide a separate language for writing tests, such as Spoofax Testing language (SPT) for Spoofax DSLs.

Editors have two main approaches to running tests: either they invoke a command-line test runner, or they provide a test runner user interface.

Just the ability to run a test suite from the command-line is enough to support testing in many code editors, including VS Code, Vim, Cloud9, and Eclipse Che. These editors start the test process by invoking a terminal command from within the user interface, and display to the user whatever the test runner prints to the standard output. The user will have to interpret the results — their formats are different for different test runners — and find the corresponding code when a test needs to be changed.

Some IDEs, such as Visual Studio, Eclipse, and IntelliJ, have a dedicated test runner user interface that displays a tree of test suites and tests, which are represented by their name and a status icon. At minimum the status icon indicates whether the test has yet to run, is currently running, was skipped, or has succeeded or failed, as shown in Figure A.42. Visual Studio can also indicate that a test was inconclusive.

The status icons of the test suites similarly show the aggregate status of their children (such as only showing success when none of the children failed). Selecting a test displays to the user the test's output, any reasons for failure, and optionally the stack trace where the failure occurred. The user can navigate from a test in the tree to the test's actual definition, and re-run a subset of tests.

A.7.1 Editor Support

Visual Studio is the only editor that has a dedicated API for a test runner. IntelliJ, while it supports a test runner, uses TeamCity messages to report the test cases and their status. Eclipse has a built-in test runner UIonly specifically for the Java language. Other languages, such as Scala, have implemented their own test runner UI.

IntelliJ

When a custom test runner prints TeamCity service messages on the standard output, IntelliJ uses these to display the status of the test cases. TeamCity is JetBrains' continuous integration software, and through the service messages the test framework can report when a (nested) test suite is started, finished, when a test is started or finished, whether a test succeeded, failed or was ignored, the test standard output and errors. (JetBrains 2017) The plugin has to provide a special run configuration which will replace the default text-based output console by a unit test console which is capable of interpreting the TeamCity messages and display the tests and their results. (JetBrains 2015b)

IntelliJ requires the test runner to be in a separate process to allow the editor to kill the testing process at the user's request, and to not impact the IDE process while the tests are running. Therefore the IntelliJ adapter has to run in this separate process as well.

The separate tester process is started by executing a command-line command with arguments, which is described by a `CommandLineState` class. The command-line state object is created by an implementation of the `RunConfiguration` interface, which manages and stores the configuration attributes. The run configuration can provide a settings form for the user to configure the run configuration, such as setting the SDK to use. (JetBrains 2018l)

A run configuration in IntelliJ is normally used to start any process, not just a test runner, and the output of the process is shown as text in the console view of the IDE. However, if the test runner outputs special TeamCity service messages that indicate the state of the tests being run, IntelliJ can recognize them and display the test results in a test results view. (JetBrains 2017) In order to make IntelliJ recognize the test runner, the `CommandLineState` object has to return an `SMTRunnerConsoleView` instead of the default text-based console view. However, this requires that the tests are run in a depth-first order. (JetBrains 2015c)

Visual Studio

A custom test runner has to implement the `ITestDiscoverer` and `ITestExecutor` interfaces, to return or execute the test cases in the current project, respectively. As shown in the API overview in Figure A.43, the test discovery implementation sends the test cases to the given `ITestCaseDiscoverySink`, whereas the test executor reports the status and results of running each test to the given `ITestExecutionRecorder`. The test result indicates, among other things, how long it took to run the test, the outcome, and optionally the error message that occurred and its corresponding stack trace. (Aniyan 2012)

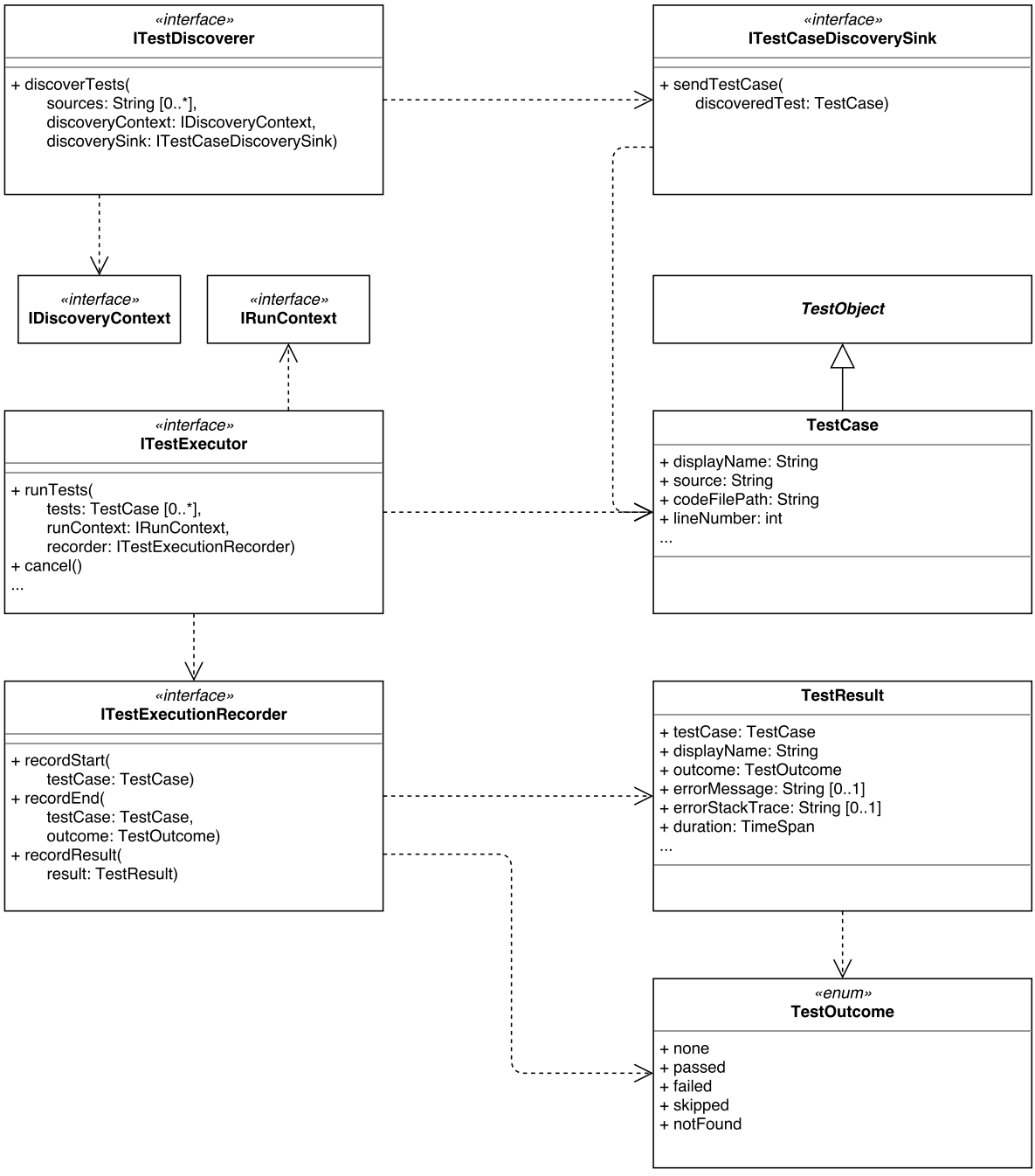


Figure A.43: Overview of the API for code completion in Visual Studio.

Appendix B

Adaptable Editor Services Interface

Chapter 5 covered the Adaptable Editor Services Interface (AESI) models for syntax coloring, code completion, structure outline, reference resolution, code actions, and debugging. In Appendices B.1 to B.7 of this chapter, we describe the models of the remaining editor services: code folding, hover documentation, signature help, automatic formatting, rename refactoring, diagnostic messages, and testing.

B.1 Code Folding

Half of the editors expose a code folding application programming interface (API). The other editors either use their syntax grammar or heuristics, or have no support for code folding at all. Most editors support character-level code folding, where the start and end of a foldable region can start at an arbitrary character within a line. A region can be assigned a default state (expanded or collapsed) and a label, and both are supported by more than one editor. In AESI we support these features as well.

The AESI model for code folding is shown in Figure B.1. The `getFoldingInfo` method of the code folding service is given a document Uniform Resource Identifier (URI) and should return a `FoldingInfo` structure which lists the foldable regions in the order they occur in that document. Each folding region, represented by a `FoldingRegion` object, specifies the part of the document that it can fold, and regions must nest properly. Optionally it can specify a label, and the default state of the region.

B.2 Hover Documentation

Hover documentation, the editor service that shows a documentation tooltip near the identifier at the caret location, is provided by the AESI `HoverDocumentationService`. The service is given a caret location and document, and returns the documentation to show and the span of code to highlight. The highlighted code shows the extent of the identifier being described by the documentation.

As Visual Studio, Eclipse, IntelliJ, Cloud9, VS Code, and Eclipse Che all support *styled* hover documentation, we support this in our model as well by using CommonMark as the styling language.

B.3 Signature Help

Signature help is similar to hover documentation, as both display extra information at the caret location, but signature help instead shows documentation for the currently selected function overload (if there are multiple) and currently active parameter.

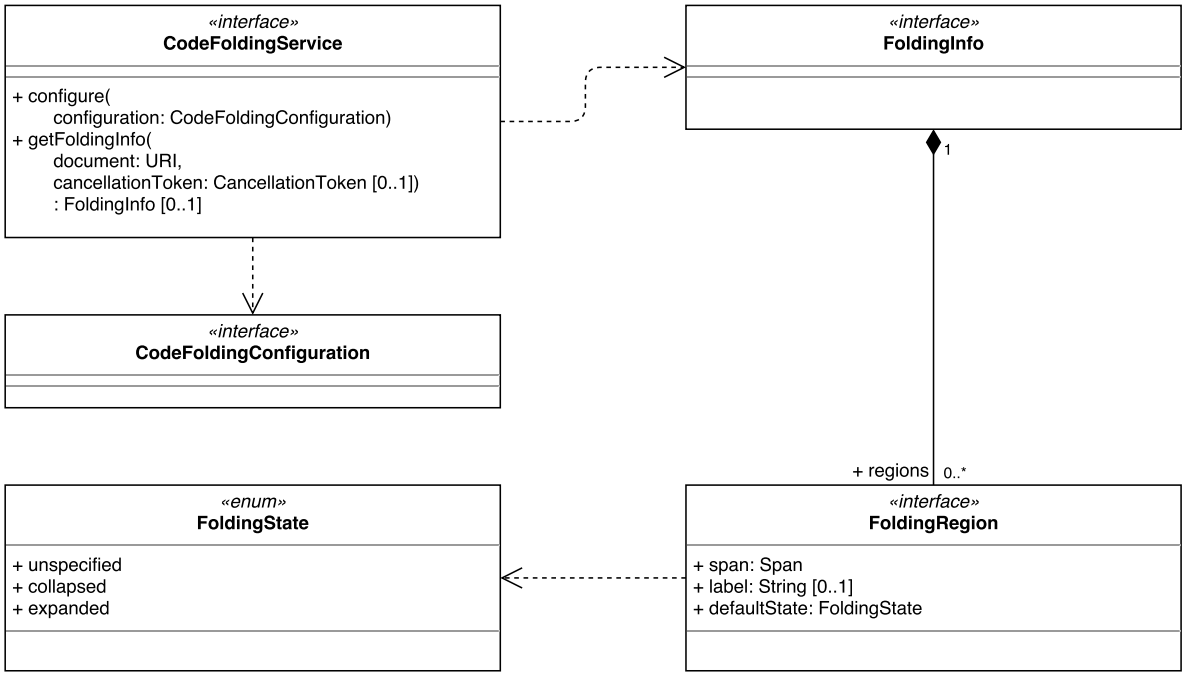


Figure B.1: Overview of the AESI API for code folding.

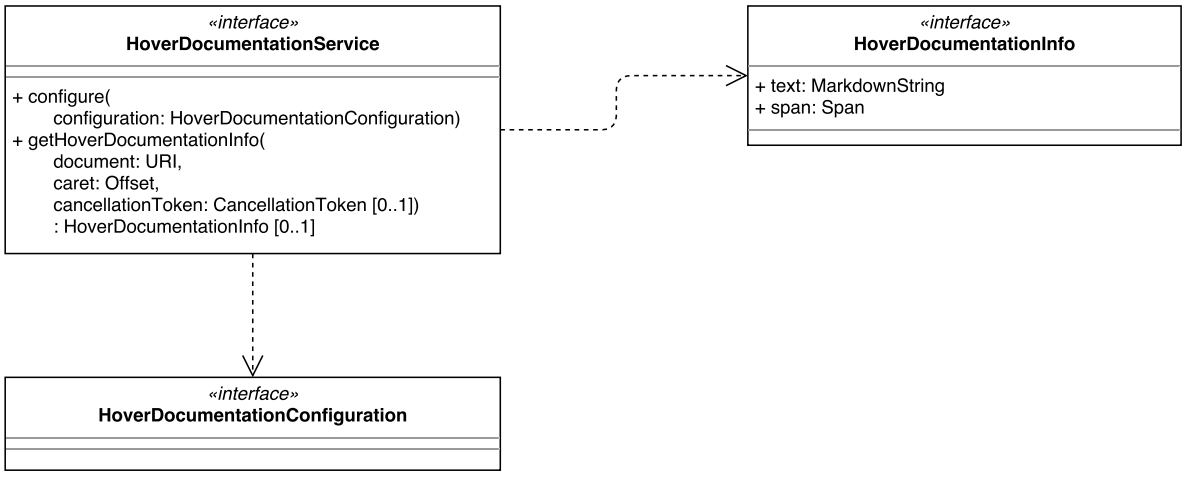


Figure B.2: Model for hover documentation.

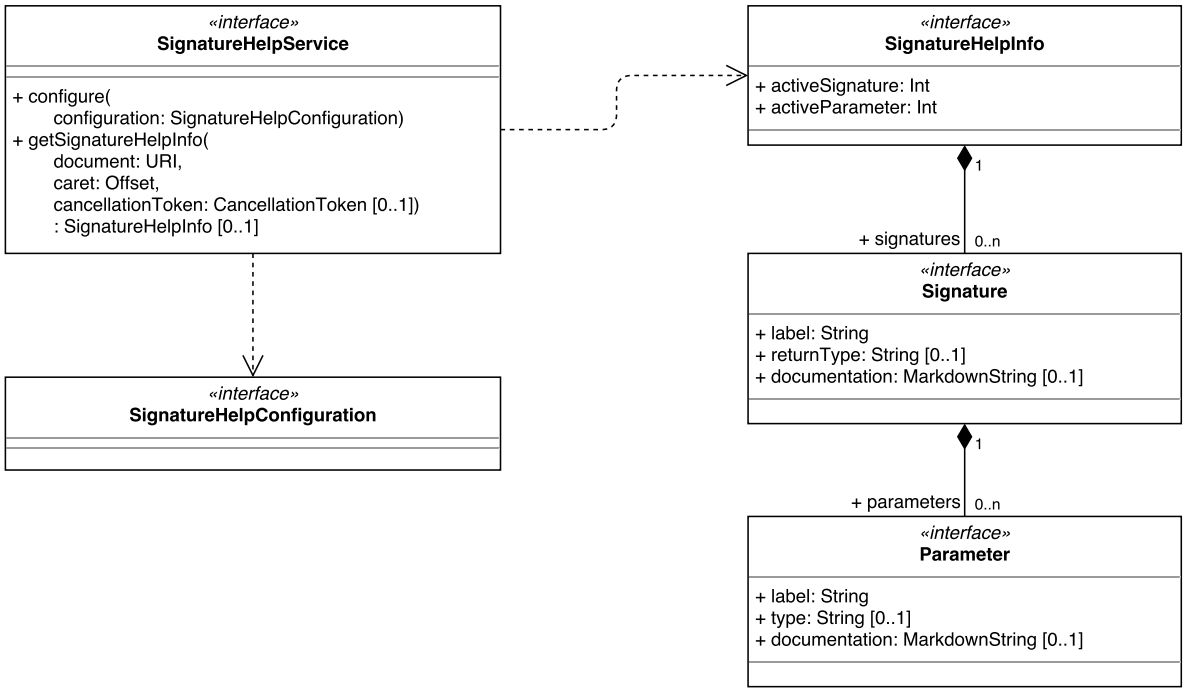


Figure B.3: Model for signature help.

The AESI signature help service, shown in Figure B.3, returns a `SignatureHelpInfo` object with the possible signatures for the function at the given caret location, where each signature has a list of parameters. Both signatures and parameters have an associated label, type, and (rich) documentation. The `SignatureHelpInfo` also specifies which parameter and signature of a function overload the editor should bring into focus, usually based on the types and number of arguments already written in the source code.

B.4 Automatic Formatting

Automatic formatting is the service that reformats the source code in a file to conform to a certain prescribed format. The service reformats a whole document, part of a document, or, in the case of live formatting, only the code just written by the user.

In our model the `FormattingService` has two main methods, shown in Figure B.4: `format` that reformats (part of) a document, and `formatOnType` that reformats the most recently written code based on the character just typed by the user. Both methods return a `FormattingInfo` object with a sequence of document edits, where each edit describes a single change to the document.

B.5 Rename Refactoring

Rename refactoring is the process that fixes all the references across documents in a project when the user changes the name of a declaration, such that the program is semantically unchanged.

When the user invokes the *rename* refactoring, most editors will show a dialog asking for the new name of the symbol at the caret location. For a good user experience, it is preferred that this dialog is not shown when there is no valid symbol name at the caret location.

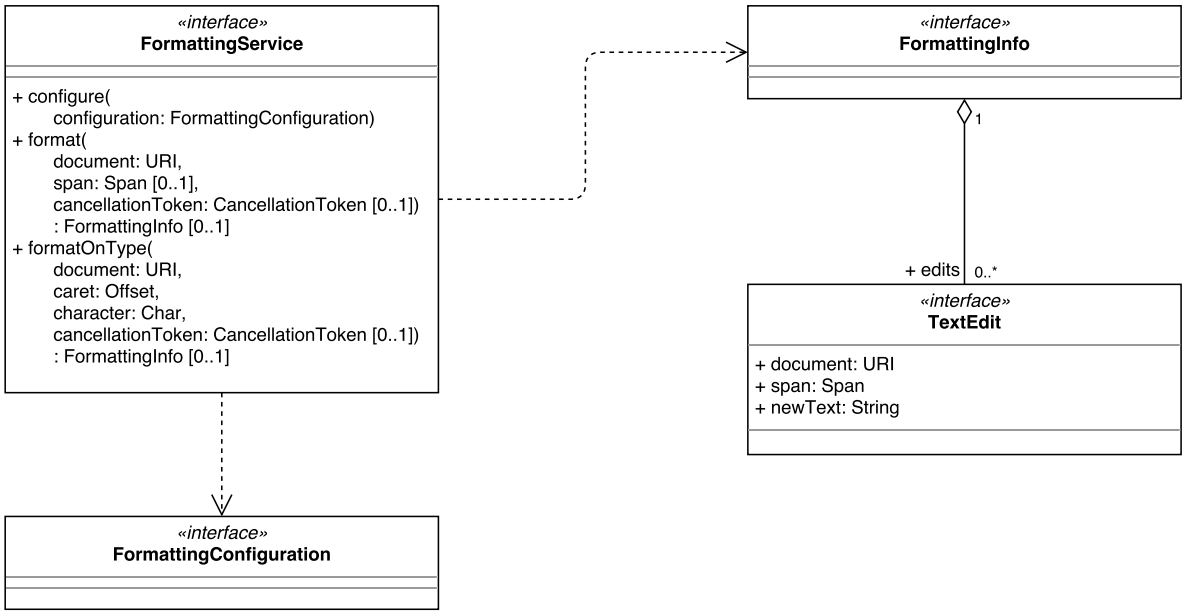


Figure B.4: Model for automatic formatting.

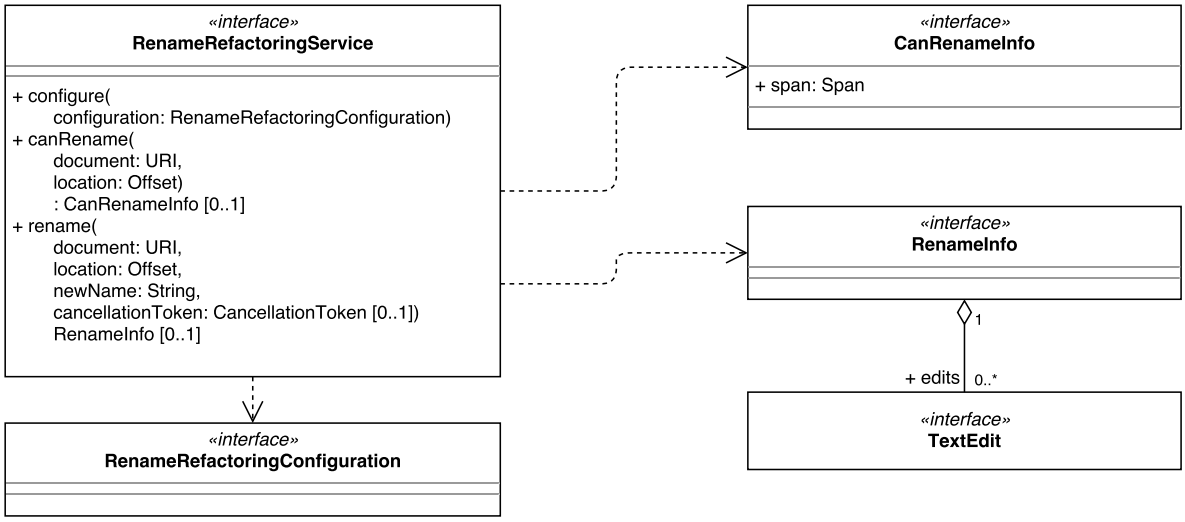


Figure B.5: Model for rename refactoring.

The `RenameRefactoringService`, shown in Figure B.5, has two primary methods: `canRename` that returns whether there is an identifier at the caret location that can be renamed, and `rename` that actually applies the rename refactoring to the project. The latter method will fail when the new name is not permitted by the language.

B.6 Diagnostic Messages

Diagnostic messages are the errors, warnings, and informational messages that are related to some part of the source code. Most editors display these messages in a list, and show a corresponding squiggly line under the affected piece of code.

A diagnostic message in AESI is represented by the `Message` interface, which has attributes

Figure B.6: Model for the message service.

for the message itself, an optional error code, the document and span to which the message applies, and the severity of the message. A message's severity indicates whether the message is an error, warning, informational message, or hint, as these four severities are best supported among editors.

Diagnostic messages are often produced while parsing or analyzing the source code, but these operations can happen as part of any editor service (such as a language implementation performing static analysis when code completion is invoked). Therefore, instead of augmenting each editor service with a way to return diagnostic messages, we opted to have a dedicated `MessageService` that is provided by the editor adapter, where editor services can control the messages being shown. The API for the service is shown in Figure B.6.

B.7 Testing

Unit tests are used to verify the correctness of the code, and many editors can run tests from within their environment and display the test results. While most editors run tests simply by invoking a command and displaying any output text to the user, only three of the editors we investigated have a dedicated testing user interface (UI) and API.

Editors display tests as a tree of test suites, with the actual tests as the leaves. The status of a test suite reflects the worst status of any of its children, such as a test suite showing failure when at least one of its tests failed. For the editor to be able to present these tests to the user, it needs the metadata of all the test cases and test suites in the project. In AESI this is provided by the test service's `getTests` method, shown in Figure B.7, which returns each test or test suite with their name and associated source location.

To run tests, the `runTests` method of the test service is given a (sub) set of test cases and a `TestReporter`. For each test the service reports back when the test is started and finished, and the result of running the test. This includes any thrown exceptions and a stack traces, and whether the test passed, failed, or was inconclusive, ignored, timed out, or aborted.

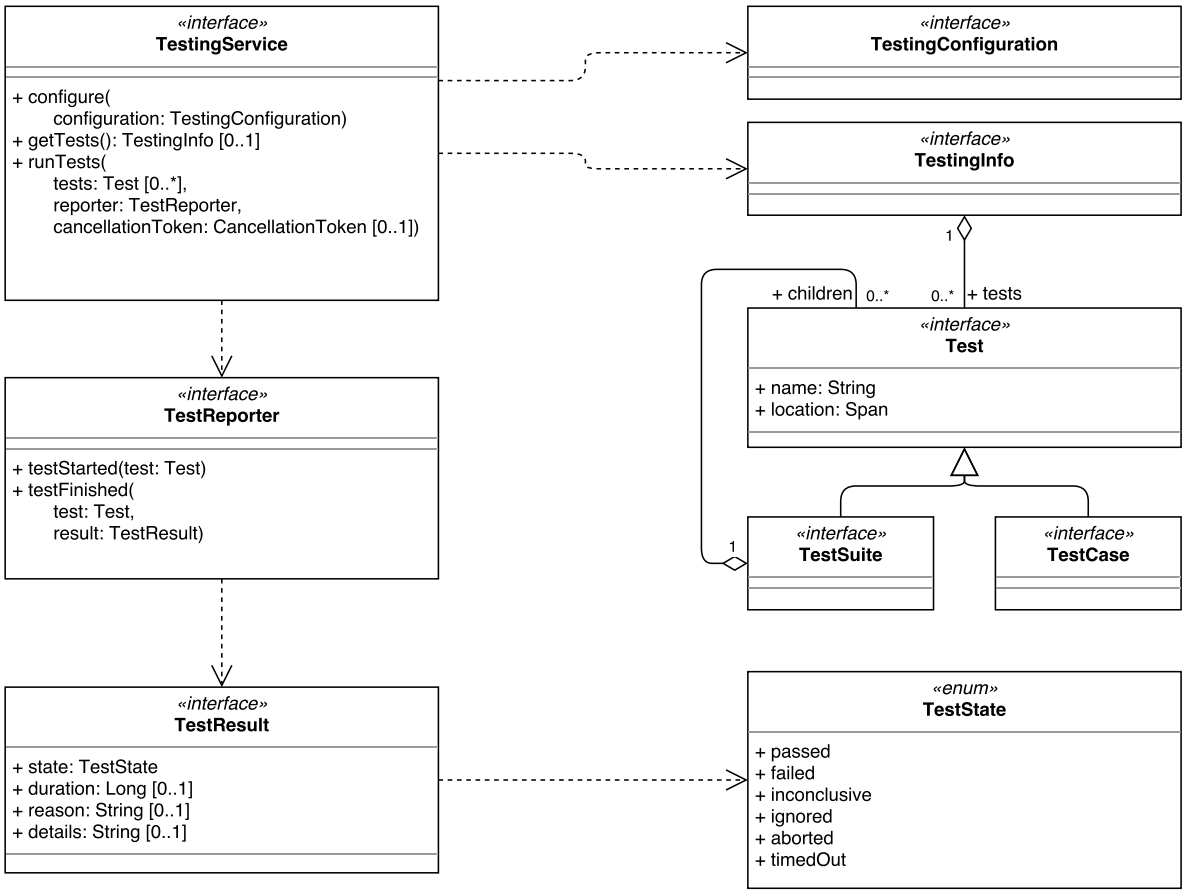


Figure B.7: Model for test discovery and execution.

Appendix C

PAPLJ Grammars

This appendix contains the full grammars we have written for the PAPLJ language in various editors. Appendix C.1 is the original grammar, from which the other grammars were derived. Appendix C.2 is the TextMate grammar as used in VS Code, Appendix C.3 has the lexer and parser grammars used in IntelliJ. Appendix C.4 is the Xtext grammar, and Appendix C.5 is the ANTLR grammar of PAPLJ.

C.1 SDF3 Grammar

Below is the original PAPLJ grammar[1], written in SDF3 by Eelco Visser as part of the book Declare Your Language (Visser 2015). The various other grammars that are part of this thesis were derived from this grammar:

```
module PAPLJ
imports Programs Classes Expressions
context-free start-symbols Program Expr
```

```
module Classes

imports Expressions Common

sorts Class Field Method Param

context-free syntax

  Class.Class = [
    class [ID] [Extends] {
      [{Field "\n"}*]
      [{Method "\n\n"}*]
    }
  ]

  Extends.NoExtends = []
  Extends.Extends = [extends [ID]]
```

[1] https://github.com/MetaBorgCube/declare-your-language/

```
  Field.Field = [[Type] [ID]]

  Method.Method =  [
    [Type] [ID]([{Param ", "}*]) [Block]
  ]

  Param.Param = [[Type] [ID]]
```

```
module Expressions

imports Common

sorts Expr
context-free syntax // arithmetic
  Expr     = [([Expr])] {bracket}
  Expr.Num = INT
  Expr.Min = [-[Expr]]
  Expr.Add = [[Expr] + [Expr]] {left}
  Expr.Sub = [[Expr] - [Expr]] {left}
  Expr.Mul = [[Expr] * [Expr]] {left}

context-free syntax // booleans
  Expr.True  = [true]
  Expr.False = [false]
  Expr.Not   = [![Expr]]
  Expr.And   = [[Expr] && [Expr]] {assoc}
  Expr.Or    = [[Expr] || [Expr]] {assoc}
  Expr.Eq    = [[Expr] == [Expr]] {non-assoc}
  Expr.Neq   = [[Expr] != [Expr]] {non-assoc}
  Expr.Lt    = [[Expr] <  [Expr]] {non-assoc}

  Expr.If = [
    if( [Expr] )
      [Expr]
    else
      [Expr]
  ] {right}

sorts Block
context-free syntax

  Expr = Block
  Block.Do = [
    {
      [{Seq ";\n"}*]
    }
  ]

  Seq = Expr
```

```
  Seq.Skip = []

context-free syntax // variables

  Expr.Var = ID

  Expr.Let = [
    let [{Bind "\n"}*]
     in [Expr]
  ]

  Bind.Bind = [[Type] [ID] = [Expr]]

context-free syntax // objects

  Expr.Get   = [[Expr].[ID]]
  Expr.Set   = [[Expr].[ID] = [Expr]]
  Expr.Call  = [[Expr].[ID]([{Expr ", "}*])]

  Expr.New   = [new [ID]()]
  Expr.This  = [this]
  Expr.Null  = [null [ID]]
  Expr.Cast  = [([ID])[Expr]]

  Type.NumT   = [Num]
  Type.BoolT  = [Bool]
  Type.UnitT  = [Unit]
  Type.ClassT = ID

context-free priorities

  {Expr.Get Expr.Call Expr.Cast}
  > {Expr.Not Expr.Min}
  > Expr.Mul
  > {left: Expr.Add Expr.Sub}
  > {non-assoc: Expr.Eq Expr.Neq Expr.Lt }
  > Expr.And
  > Expr.Or
  > Expr.Set
  > {Expr.If Expr.Let}

template options

  keyword -/- [a-zA-Z0-9]

lexical syntax // reserved words
  ID      = Keyword {reject}
  Keyword = "Num"
  Keyword = "Bool"
  Keyword = "true"
  Keyword = "false"
  Keyword = "this"
```

```
Keyword = "new"
```

```
module Programs

imports Classes Expressions Common

sorts Program

context-free syntax

  Program.Program = [
    program [ProgramName]

    [{Class "\n\n"}*]

    run
      [Expr]
  ]
```

```
module Values

imports Common Classes

context-free syntax

  Value.NumV   = INT
  BoolV.TrueV  = <true>
  BoolV.FalseV = <false>
  Value.BoolV  = BoolV
  ObjV.ObjV    = <obj(<ID>, <SuperObj>, <Map>, <Map>)>
  Value.o2v    = ObjV

  SuperObj.NoSuper = <>
  SuperObj.Super = ObjV

  Map.Map = <<ID>{<{Bind ","}*>}>
  Bind.Bind = [[ID] |--> [Value]]
```

```
module Common
lexical syntax
  ProgramName    = [a-zA-Z0-9\-\_]*
  ID             = [a-zA-Z] [a-zA-Z0-9\_]*
  INT            = [\-]? [0-9]+
```

```
  STRING         = "\"" StringChar* "\""
  StringChar     = ~[\"\n]
  StringChar     = "\\\""
  StringChar     = BackSlashChar
  BackSlashChar  = "\\"
  LAYOUT         = [\ \t\n\r]
  CommentChar    = [\*]
  LAYOUT         = "/*" InsideComment* "*/"
  InsideComment  = ~[\*]
  InsideComment  = CommentChar
  LAYOUT         = "//" ~[\n\r]* NewLineEOF
  NewLineEOF     = [\n\r]
  NewLineEOF     = EOF
  EOF            =

lexical restrictions
  // Ensure greedy matching for lexicals
  CommentChar   -/- [\/]
  INT           -/- [0-9]
  ID            -/- [a-zA-Z0-9\_]
  "-"           -/- [0-9]

  // EOF may not be followed by any char
  EOF           -/- ~[]

  // Backslash chars in strings may not be followed by "
  BackSlashChar -/- [\"]

context-free restrictions
  // Ensure greedy matching for comments
  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [\/].[\/]
LAYOUT? -/- [\/].[\*]
```

C.2 TextMate Grammar

Below is the TextMate grammar of the PAPLJ language as used in the VS Code plugin.

```
{
  "$schema": "https://raw.githubusercontent.com/martinring/tmlanguage/master/tmlanguage.json",
  "name": "PAPLJ",
  "scopeName": "source.paplj",
  "fileTypes": [ "pj" ],

  "foldingStartMarker": "(\\{|\\()\\s*$",
  "foldingStopMarker": "(\\}|\\))\\s*$",

  "patterns": [
    { "include": "#comments" },
    {
      "name": "meta.program.paplj",
      "comment": "Matches the `'program' ID (';')?` statement.",
      "begin": "^\\s*(program)\\b\\s*",
      "beginCaptures": {
        "1": { "name": "keyword.other.program.paplj" }
      },
      "contentName": "storage.modifier.program.paplj",
      "patterns": [
        { "include": "#qualified_id" }
      ],
      "end": "\\s*(?:$|(;))",
      "endCaptures": {
        "1": { "name": "punctuation.terminator.paplj" }
      }
    },
    {
      "name": "meta.import.paplj",
      "comment": "Matches the `'import' ID (';')?` statement.",
      "begin": "(import)\\b\\s*",
      "beginCaptures": {
        "1": { "name": "keyword.other.import.paplj" }
      },
      "contentName": "storage.modifier.import.paplj",
      "patterns": [
        { "include": "#qualified_id" }
      ],
      "end": "\\s*(?:$|(;))",
      "endCaptures": {
        "1": { "name": "punctuation.terminator.paplj" }
      }
    },
    { "include": "#strings" },
    { "include": "#numeric" },
    { "include": "#constants" },
    { "include": "#brackets" },
    { "include": "#delimiters" },
```

```
    { "include": "#operators" },
    { "include": "#keywords" },
    { "include": "#storage_types" },
    { "include": "#identifiers" }
  ],
  "repository": {
    "comments": {
      "patterns": [
        {
          "name": "comment.block.paplj",
          "comment": "Matches a block comment between /* and */.",
          "begin": "/\\*",
          "end": "\\*/",
          "captures": {
            "0": {
              "name": "punctuation.definition.comment.paplj"
            }
          }
        },
        {
          "name": "comment.line.double-slash.paplj",
          "comment": "Matches a line comment that starts with //.",
          "begin": "//",
          "beginCaptures": {
            "0": {
              "name": "punctuation.definition.comment.paplj"
            }
          },
          "end": "$"
        }
      ]
    },
    "qualified_id": {
      "patterns": [
        {
          "name": "punctuation.separator.paplj",
          "match": "\\."
        },
        {
          "name": "invalid.illegal.character_not_allowed_here.paplj",
          "match": "\\s"
        }
      ]
    },
    "strings": {
      "patterns": [
        {
          "name": "string.quoted.double.paplj",
          "comment": "Matches a double quoted string.",
          "begin": "\"",
          "beginCaptures": {
            "0": {
```

```
            "name": "punctuation.definition.string.begin.paplj"
          }
        },
        "patterns": [
          {
            "name": "constant.character.escape.paplj",
            "comment": "Matches \\r, \\n, \\t, \\\\, \\', \\\".",
            "match": "\\\\[rnt\\\\'\"]"
          },
          {
            "name": "invalid.illegal.unknown-escape.paplj",
            "comment": "Marks other character escapes as illegal.",
            "match": "\\\\[^rnt\\\\'\"]"
          }
        ],
        "end": "\"",
        "endCaptures": {
          "0": {
            "name": "punctuation.definition.string.end.paplj"
          }
        }
      }
    ]
  },
  "numeric": {
    "patterns": [
      {
        "name": "constant.numeric.integer.paplj",
        "comment": "Matches integer literals.",
        "match": "\\b\\d+\\b"
      }
    ]
  },
  "constants": {
    "patterns": [
      {
        "name": "constant.language.paplj",
        "comment": "Matches language constants.",
        "match": "\\b(true|false|this|null)\\b"
      }
    ]
  },
  "brackets": {
    "patterns": [
      {
        "name": "punctuation.other.bracket.curly.paplj",
        "match": "\\{|\\}"
      },
      {
        "name": "punctuation.other.bracket.round.paplj",
        "match": "\\(|\\)"
      },
```

```
      {
        "name": "punctuation.other.bracket.square.paplj",
        "match": "\\[|\\]"
      },
      {
        "name": "punctuation.other.bracket.angled.paplj",
        "match": "\\<|\\>"
      }
    ]
  },
  "delimiters": {
    "patterns": [
      {
        "name": "punctuation.other.comma.paplj",
        "match": ","
      }
    ]
  },
  "operators": {
    "comment": "NOTE: The order matters.",
    "patterns": [
      {
        "name": "keyword.operator.comparison.paplj",
        "match": "(==|!=|<)"
      },
      {
        "name": "keyword.operator.logical.paplj",
        "match": "(&&|\\|\\||!)"
      },
      {
        "name": "keyword.operator.assignment.paplj",
        "match": "(=)"
      },
      {
        "name": "keyword.operator.arithmetic.paplj",
        "match": "(\\+|\\-|\\*|/)"
      }
    ]
  },
  "keywords": {
    "patterns": [
      {
        "name": "keyword.control.paplj",
        "match": "\\b(if|else|let|in|new)\\b"
      },
      {
        "name": "keyword.other.paplj",
        "match": "\\b(program|run|class|extends|as|public|import)\\b"
      }
    ]
  },
  "storage_types": {
```

```
      "patterns": [
        {
          "name": "storage.type.numeric.paplj",
          "match": "\\bNum\\b"
        },
        {
          "name": "storage.type.boolean.paplj",
          "match": "\\bBool\\b"
        },
        {
          "name": "storage.type.unit.paplj",
          "match": "\\bUnit\\b"
        },
        {
          "name": "storage.type.any.paplj",
          "match": "\\bAny\\b"
        },
        {
          "name": "storage.type.null.paplj",
          "match": "\\bNull\\b"
        }
      ]
    },
    "identifiers": {
      "patterns": [
        {
          "name": "variable.name.paplj",
          "match": "\\b[a-zA-Z]\\w*\\b"
        }
      ]
    }
  }
}
```

C.3 IntelliJ Grammars

Below are the lexer parser grammars for the PAPLJ language as used in the IntelliJ plugin.

The JFlex lexer grammar:

```
package org.metaborg.paplj;

import com.intellij.lexer.FlexLexer;
import com.intellij.psi.tree.IElementType;

import static com.intellij.psi.TokenType.BAD_CHARACTER;
import static com.intellij.psi.TokenType.WHITE_SPACE;
import static org.metaborg.paplj.psi.PapljTypes.*;
import static org.metaborg.paplj.psi.PapljTokenElementTypes.*;

%%

%{
  public _PapljLexer() {
    this((java.io.Reader)null);
  }
%}

%public
%class _PapljLexer
%implements FlexLexer
%function advance
%type IElementType
%unicode

EOL=\R
WHITE_SPACE=\s+

ID=[:letter:][a-zA-Z_0-9]*
INT=[0-9]+

%%
<YYINITIAL> {
  {WHITE_SPACE}      { return WHITE_SPACE; }

  "#"                { return SHA; }
  "!"                { return EXCL; }
  ";"                { return SEMICOLON; }
  "."                { return DOT; }
  "*"                { return ASTERISK; }
  "+"                { return PLUS; }
  "-"                { return MINUS; }
  "/"                { return SLASH; }
  "("                { return BRACE_L; }
  ")"                { return BRACE_R; }
  "{"                { return CBRACE_L; }
  "}"                { return CBRACE_R; }
```

```
  "["                { return BRACK_L; }
  "]"                { return BRACK_R; }
  "<"                { return ABRACK_L; }
  ">"                { return ABRACK_R; }
  ","                { return COMMA; }
  "="                { return EQ; }
  "||"               { return OROR; }
  "&&"               { return ANDAND; }
  "=="               { return EQEQ; }
  "!="               { return EXCLEQ; }
  "public"           { return K_PUBLIC; }
  "import"           { return K_IMPORT; }
  "program"          { return K_PROGRAM; }
  "class"            { return K_CLASS; }
  "extends"          { return K_EXTENDS; }
  "run"              { return K_RUN; }
  "true"             { return K_TRUE; }
  "false"            { return K_FALSE; }
  "new"              { return K_NEW; }
  "null"             { return K_NULL; }
  "this"             { return K_THIS; }
  "if"               { return K_IF; }
  "else"             { return K_ELSE; }
  "let"              { return K_LET; }
  "in"               { return K_IN; }
  "as"               { return K_AS; }

  {ID}               { return ID; }
  {INT}              { return INT; }

}

[^] { return BAD_CHARACTER; }
```

The Backus–Naur form (BNF) parser grammar:

```
{
  parserClass="org.metaborg.paplj.parser.PapljParser"
  parserUtilClass="org.metaborg.paplj.parser.PapljParserUtil"
  parserImports=["static org.metaborg.paplj.psi.PapljTokenElementTypes.*"]

  implements="org.metaborg.paplj.psi.PapljCompositeElement"
  extends="org.metaborg.paplj.psi.impl.PapljCompositeElementImpl"

  elementTypeHolderClass="org.metaborg.paplj.psi.PapljTypes"
  elementTypeClass="org.metaborg.paplj.psi.PapljCompsiteElementType"

  tokenTypeClass="org.metaborg.paplj.psi.PapljTokenType"

  psiClassPrefix="Paplj"
  psiImplClassSuffix="Impl"
  psiPackage="org.metaborg.paplj.psi"
```

```
psiImplPackage="org.metaborg.paplj.psi.impl"

generateTokens=false
generateTokenAccessors=true

tokens = [
  SHA                             = '#'
  EXCL                            = '!'
  SEMICOLON                       = ';'
  DOT                             = '.'
  ASTERISK                        = '*'
  PLUS                            = '+'
  MINUS                           = '-'
  SLASH                           = '/'
  BRACE_L                         = '('
  BRACE_R                         = ')'
  CBRACE_L                        = '{'
  CBRACE_R                        = '}'
  BRACK_L                         = '['
  BRACK_R                         = ']'
  ABRACK_L                        = '<'
  ABRACK_R                        = '>'
  COMMA                           = ','
  EQ                              = '='
  ORAND                           = '||'
  ANDAND                          = '&&'
  EQEQ                            = '=='
  EXCLEQ                          = '!='

  K_PUBLIC                        = 'public'
  K_IMPORT                        = 'import'
  K_PROGRAM                       = 'program'
  K_CLASS                         = 'class'
  K_EXTENDS                       = 'extends'
  K_RUN                           = 'run'
  K_TRUE                          = 'true'
  K_FALSE                         = 'false'
  K_NEW                           = 'new'
  K_NULL                          = 'null'
  K_THIS                          = 'this'
  K_IF                            = 'if'
  K_ELSE                          = 'else'
  K_LET                           = 'let'
  K_IN                            = 'in'
  K_AS                            = 'as'

  ID                              = 'regexp:\p{Alpha}\w*'
  INT                             = 'regexp:\d+'
]

extends(".*_expr")=expr
```

```
}

file                        ::= programStatement?
                                importStatement*
                                classDeclaration*
                                runStatement?

programStatement            ::= 'program' qualifiedID ';'?

importStatement             ::= 'import' wildcardID ';'?

classDeclaration            ::= 'class' ID ('extends' qualifiedID)?
                                '{' memberDeclaration* '}'

memberDeclaration           ::= method | field

method                      ::= qualifiedID ID '(' (param (',' param)*)? ')'
                                block_expr

field                       ::= qualifiedID ID ';'?

param                       ::= qualifiedID ID

runStatement                ::= 'run' expr

expr                        ::= if_expr | let_expr | assignment

if_expr                     ::= 'if' '(' expr ')' expr 'else' expr

let_expr                    ::= 'let' binding* 'in' expr

binding                     ::= qualifiedID ID '=' expr

private assignment          ::= logical_or assignment_expr?
left assignment_expr        ::= '=' logical_or

private logical_or          ::= logical_and logical_or_expr*
left logical_or_expr        ::= '||' logical_and

private logical_and         ::= comparative logical_and_expr*
left logical_and_expr       ::= '&&' comparative

private comparative         ::= additive comparative_expr?
left comparative_expr       ::= ('==' | '!=' | '<') additive

private additive            ::= multiplicative additive_expr*
left additive_expr          ::= ('+' | '-') multiplicative

private multiplicative      ::= unary multiplicative_expr*
left multiplicative_expr    ::= ('*' | '/') unary*

private unary               ::= not_expr | min_expr | (member cast_expr?)
```

```
left cast_expr              ::= 'as' qualifiedID
not_expr                    ::= '!' unary
min_expr                    ::= '-' unary

private member              ::= primary member_expr*
left member_expr            ::= '.' ID ('(' (expr (',' expr)*)? ')')?

private primary             ::= num_expr | bool_expr | this_expr | null_expr
                              | new_expr | var_expr | block_expr | paren_expr
num_expr                    ::= INT
bool_expr                   ::= 'true' | 'false'
this_expr                   ::= 'this'
null_expr                   ::= 'null' qualifiedID?
new_expr                    ::= 'new' qualifiedID '(' ')'
var_expr                    ::= ID ('(' (expr (',' expr)*)? ')')?
block_expr                  ::= '{' (expr (';' expr)*)? ';'? '}'
private paren_expr          ::= '(' expr ')'

private qualifiedID         ::= ID ('.' ID)*

private wildcardID          ::= ID ('.' ID)* ('.' '*')?
```

C.4 Xtext Grammar

Below is the Xtext grammar of PAPLJ as used in the Xtext PAPLJ plugin.

```
grammar org.metaborg.paplj.Paplj with org.eclipse.xtext.common.Terminals

generate paplj "http://www.metaborg.org/paplj/Paplj"

////////////////////////
// Program and Classes //
////////////////////////

// The first rule is the start rule, and must be a non-terminal.
Program:
  ('program' name=QualifiedName ';'?)?

  imports+=Import*

    classes+=Type*

    ('run'
        expr=Expr)?
;

QualifiedName: ID ('.' ID)*;
QualifiedNameWithWildcard: QualifiedName '.*'?;

Import: 'import' importedNamespace=QualifiedNameWithWildcard ';'?;

Type:
  'class' name=ID ('extends' superType=[Type|QualifiedName])? '{'
    members+=Member*
  '}'
;

Member: Field | Method;
Symbol: Param | Binding | Member;

Field: type=[Type|QualifiedName] name=ID ';'?;

Method: type=[Type|QualifiedName] name=ID '(' (params+=Param (',' params+=Param)*)? ')'
    body=Block2
;

Param: type=[Type|QualifiedName] name=ID;

Binding: type=[Type|QualifiedName] name=ID '=' value=Expr;

Block2: {Block2}
    '{'
        (exprs+=Expr (';' exprs+=Expr)*)? ';'?
    '}'
```

```
;


//////////////////
// Expressions //
//////////////////

// From lowest to highest precedence.
Expr returns Expr:
  IfLetExpr
;

IfLetExpr returns Expr:
    If |
    Let |
    AssignmentExpr
;

// Non-associative.
AssignmentExpr returns Expr:
  LogicalOrExpr (
    =>({Assignment.left=current} '=' value=Expr)
  )?
;

// Left-associative.
LogicalOrExpr returns Expr:
    LogicalAndExpr ((
            {Or.left=current}  '||'
    ) right=LogicalAndExpr)*
;

// Left-associative.
LogicalAndExpr returns Expr:
    ComparativeExpr ((
            {And.left=current} '&&'
    ) right=ComparativeExpr)*
;

// Non-associative.
ComparativeExpr returns Expr:
    AdditiveExpr ((
            {Eq.left=current}  '==' |
            {Neq.left=current} '!=' |
            {Lt.left=current}  '<'
    ) right=AdditiveExpr)?
;

// Left-associative.
AdditiveExpr returns Expr:
    MultiplicativeExpr ((
            {Add.left=current} '+' |
```

```
            {Sub.left=current} '-'
    ) right=MultiplicativeExpr)*
;

// Left-associative.
MultiplicativeExpr returns Expr:
    UnaryExpr ((
            {Mul.left=current} '*' |
            {Div.left=current} '/'
    ) right=UnaryExpr)*
;

// Non-associative.
UnaryExpr returns Expr:
    MemberExpr ((
            {Cast.left=current} 'as'
    ) type=[Type])? |
    {Not} '!' expr=UnaryExpr |
    {Min} '-' expr=UnaryExpr
;

// Left-associative.
MemberExpr returns Expr:
  PrimaryExpr (
    =>({MemberRef.left=current} '.' member=[Member] (
      (methodInvocation?='(' (args+=Expr (',' args+=Expr)*)? ')')?
    ))
  )*
;

PrimaryExpr returns Expr:
    {Num} value=INT |
    {Bool} (^true?='true' | 'false') |
    {This} 'this' |
    {Null} 'null' (type=[Type])? |
    {New} 'new' type=[Type|QualifiedName] '(' ')' |
    {Var} member=[Symbol] (methodInvocation?='(' (args+=Expr (',' args+=Expr)*)? ')')? |
    Block2 |
    '(' Expr ')'
;

////////////////////////////////////////////////////////////////

If:
    'if' '(' condition=Expr ')'
        onTrue=Expr
    'else'
        onFalse=Expr
;

Let:
    'let' (bindings+=Binding)*
```

```
    'in' expr=Expr
;
```

C.5 ANTLR Grammars

Below are the ANTLR4 lexer and parser grammars as used in the AESI implementation.

The lexer grammar:

```
lexer grammar PapljAntlrLexer;

@header {
package com.virtlink.paplj.syntax;
}
channels {
    WHITESPACE,
    COMMENTS
}

PROGRAM     : 'program';
RUN         : 'run';
IMPORT      : 'import';
CLASS       : 'class';
EXTENDS     : 'extends';
IF          : 'if';
ELSE        : 'else';
LET         : 'let';
IN          : 'in';
AS          : 'as';
TRUE        : 'true';
FALSE       : 'false';
THIS        : 'this';
NULL        : 'null';
NEW         : 'new';


SEMICOLON   : ';';
DOTSTAR     : '.*';
COMMA       : ',';
DOT         : '.';
LBRACE      : '{';
RBRACE      : '}';
LPAREN      : '(';
RPAREN      : ')';
EQ          : '==';
NEQ         : '!=';
LTE         : '<=';
GTE         : '>=';
LT          : '<';
GT          : '>';
OR          : '||';
AND         : '&&';
ASSIGN      : '=';
PLUS        : '+';
MIN         : '-';
MUL         : '*';
```

```
DIV         : '/';
NOT         : '!';

ID          : [A-Za-z]+;
INT         : [0-9]+;
COMMENT     : '/*' .*? '*/' -> channel(COMMENTS);
LINE_COMMENT: '//' .*? '\r'? '\n' -> channel(COMMENTS);
WS          : [ \t\r\n]+ -> channel(WHITESPACE);
```

The parser grammar:

```
parser grammar PapljAntlrParser;
@header {
package com.virtlink.paplj.syntax;
}

options { tokenVocab=PapljAntlrLexer; }

////////////////////////
// Program and Classes //
////////////////////////

// The first rule is the start rule, and must be a non-terminal.
program :
    'program' qualifiedName ';'?

    imports*
    type*

    ('run' expr ';'?)?
;

qualifiedName: ID ('.' ID)* ;
qualifiedNameWithWildcard: qualifiedName '.*'?;

imports: 'import' qualifiedNameWithWildcard ';'?;

type:
    'class' ID ('extends' qualifiedName) '{'
        classMember*
    '}'
;

classMember: field | method;

field: qualifiedName ID ';'?;

method: qualifiedName ID '(' (param (',' param)*)? ')'
    block2
;

param: qualifiedName ID;
```

```
binding: qualifiedName ID '=' expr;

block2:
    '{'
        (expr (';' expr)*)? ';'?
    '}'
;



/////////////////
// Expressions //
/////////////////

expr
    : '(' expr ')'                              # Parens
    | block2                                    # Block
    | ID                                        # Var
    | ID '(' (expr (',' expr)*)? ')'            # Call
    | 'new' qualifiedName '(' ')'               # New
    | 'null' qualifiedName?                     # Null
    | 'this'                                    # This
    | v=('true' | 'false')                      # Bool
    | INT                                       # Num
    | expr '.' ID                               # Member
    | expr '.' ID '(' (expr (',' expr)*)? ')'   # MemberCall
    | '-' expr                                  # Negate
    | '!' expr                                  # Not
    | <assoc=right> expr 'as' qualifiedName     # Cast
    | expr op=('*' | '/') expr                  # Multiplicative
    | expr op=('+' | '-') expr                  # Additive
    | expr op=('==' | '!=' | '<') expr          # Compare
    | expr '&&' expr                            # And
    | expr '||' expr                            # Or
    | <assoc=right> expr '=' expr               # Assignment
    | 'let' binding* 'in' expr                  # Let
    | 'if' '(' expr ')' expr 'else' expr        # If
;
```

Appendix D

Snippet Syntax

A code snippet in AESI, which can be inserted using the code completions service from Section 5.4, has the following syntax:

```
any         ::= tabstop
              | placeholder
              | variable
              | text
tabstop     ::= '$' int
              | '${' int '}'
placeholder ::= '${' int ':' any '}'
variable    ::= '$' var
              | '${' var }'
              | '${' var ':' any '}'
var         ::= [_a-zA-Z] [_a-zA-Z0-9]*
int         ::= [0-9]+
text        ::= .*
```

The dollar-sign (`$`), left-curly bracket (`{`), and back-slash (`\`) special characters can be escaped by directly preceding them by a back-slash (`\`).