HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

TypeScript Refactorings

Bachelor Thesis

University of Applied Sciences Rapperswil

Fall 2018

Authors:	Giovanni Heilmann, Arooran Thanabalasingam
Advisors:	Thomas Corbat, Felix Morgner
Industry Partner:	Institute for Software (IFS) HSR
Co-Examiner:	Lukas Felber
Co-Reader:	Prof. Dr. Farhad Mehta

Assignment

Supervisor and Expert

This bachelor thesis will be conducted with the Institute for Software at HSR. It will be supervised by Thomas Corbat (tcorbat@hsr.ch) and Felix Morgner (fmorgner@hsr.ch), HSR, IFS. An expert independent of HSR will examine the thesis and will be present at the final presentation:

- Lukas Felber, Quatico Solutions AG (lukas.felber@quatico.com)

Student

This project is conducted in the context of the module bachelor thesis at the department of computer science by

- Giovanni Heilmann (gheilman@hsr.ch)
- Arooran Thanabalasingam (athanaba@hsr.ch)

Introduction

TypeScript is a programming language developed by Microsoft [35]. The development environment Visual Studio Code features a builtin plug-in with semantic support for TypeScript development. Among other functionality, it supports debugging, offers syntax highlighting and code actions like refactorings and quick fixes [21]. The plug-in uses the TypeScript language server to provide these features. The TypeScript language server is an open source project, hosted on Github [33] and external contributions are possible.
Refactoring is a controlled technique to improve the quality of an existing code base without changing its observable behavior [13]. It is an iterative process heavily relying on good test coverage of the code to be refactored. A quick fix is an immediate solution for a transient error or deficiency in the code while programming. Proper tooling support for both features automates and accelerates repetitive coding tasks. Besides productivity, such automated code changes are less error prone than the same modifications applied manually.

Goals of the Project

Visual Studio Code currently offers only a few code actions for TypeScript. The goal of this project is to extend the TypeScript language server with additional refactorings and quick fixes. This extension affects two components of the plug-in:

- User interface for interaction with the developer.
- TypeScript language server, which contains all capabilities for semantic analysis of the source code of a project. It also features infrastructure for rendering the actual code transformations based on modifications of the abstract syntax tree.

In a preceding analysis of the features already available, additional code actions to be implemented had been identified. The potential features are recorded on the TypeScript's issue tracker on GitHub [34]. The selected items can be seen in table 1. Based on usefulness to the TypeScript community those have been prioritized. It contains the corresponding issue number for GitHub.
It is estimated that for successful completion at least the following code actions can be completed:

- Interface stubbing
- Inline local variable
- Lambda to function
- Inline method

A feature is considered complete, when a ready-to-be-merged pull request exists for the TypeScript plug-in repository. Since the actual integration is not in the hands of the students and cannot be influenced directly by HSR, it is not a requirement.

If the team progresses faster than the schedule, further code actions will be implemented. There will be weekly meetings with the supervisors. Additional meetings might be scheduled as required by the students. All Meetings, except for the kick-off meeting, will be prepared by the students with an agenda that is sent to the supervisors at least one day before the meeting. During the meeting the current progress will be presented (What has been done? What has been achieved? How much time did it take? What is planned for the subsequent week?). Decisions of the meeting must be recorded by the students.

At the beginning a project plan has to be devised for with milestones for the semester. This plan is used as a guide line to check the progress compared to the

estimation. The project plan will be updated according to the actual execution, including time reports and tasks. The students get feedback for accomplished milestones. The final mark will be given based on the eventual results handed-in by the deadline at the end of the semester.
In a concluding presentation, the students will demonstrate the features they have implemented and discuss their results.

Issue No	**Description**	**Type**	**Priority**
#16755	Interface stubbing	Quick-Fix	1
#18459	Inline local variable	Refactoring	1
#23299	Lambda to function	Refactoring	1
#27070	Inline method	Refactoring	1
#662	Reorder parameters	Refactoring	2
#18267	String concatenation -> template literals	Refactoring	2
#24827	Destructure function parameters	Refactoring	2
	Move method to module/namespace	Refactoring	2
#16010	File capitalization	Quick-Fix	3
#22392	Create object from selected variables	Refactoring	3
#23552	Named parameter	Refactoring	3
#23830	Auto-import module as namespace	Quick-Fix	3
#23869	Type alias	Refactoring	3
#25175	Logic predicate	Refactoring	3
#25946	Var to destructuring	Refactoring	3
#26479	Extract function to outer scope	Refactoring	3
#26487	Merge imports from same source	Refactoring	3

Table 1.: Prioritized issues

Documentation

This project must be documented according to the guide lines of the department of computer science [2]. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server [3]. Two copies of the documentation must be handed-in:

- One in color, two-sided
- One in B/W, single-sided

Important Dates

17.09.2018	Start of bachelor thesis
Until 18.12.2018	Hand-in of the abstract to the supervisor for checking (on abstract.hsr.ch). Information about accessing the corresponding web tool will be given by the department office. Hand-in (by email) of the A0 poster to the supervisor.
21.12.2018, 12.00	Final hand-in of the report through archiv-i.hsr.ch
08.01.2019	Presentation and Oral Exam

Table 2.: Important dates

Evaluation

A successful bachelor thesis counts as 12 ECTS points per student. The estimated effort for 1 ECTS is 30 hours. (See also the module description [27]). The supervisor will be in charge for all the evaluation of the project.

Criterion	**Weighting**
1. Organisation, Execution	$\frac{1}{6}$
2. Report (Abstract, Management Summary, technical and personal reports) as well as structure, visualization and language of the whole documentation	$\frac{1}{6}$
3. Content	$\frac{3}{6}$
4. Final Presentation of the results and discussion	$\frac{1}{6}$

Table 3.: Evaluation criteria

Furthermore, the bachelor thesis must adhere to the general regulations of the department of computer science.

Rapperswil, 17th September 2018

Thomas Corbat

Lecturer
Institut für Software (IFS)
Hochschule für Technik Rapperswil

Abstract

Visual Studio Code is a text editor written in TypeScript, which is a superset of JavaScript with strong typing. Language support and other features can be added via extensions. Both TypeScript and VS Code are open-source and maintained by Microsoft. The current TypeScript implementation has in-built language features like auto-complete, reference lookup, etc. Two such features are refactorings and quick-fixes. Refactorings are behavior-preserving code changes that make code clearer and more maintainable. Quick-fixes are code changes that change external behavior. They fix mistakes or extend existing behavior.
Currently, VS Code lacks a lot of the basic refactorings and quick-fixes for TypeScript like *inline variable* or *inline function*. Some of the few refactorings it supports, are *Extract symbol* and *Move to new file*. The goal of this bachelor thesis is to contribute some of the missing refactorings and quick-fixes to the TypeScript repository.
In total, four refactorings have been implemented. First, *Inline local* replaces one or all references of a local variable with its value. *Inline function* is similar to the first, but it inlines a function declaration. Third, *Convert lambda to function* converts an arrow function to a function or vice-versa without changing external behavior. Lastly, *Convert string concatenation to template literal* converts a concatenated string like `"hello "+ world` to a template literal like `` `hello ${world}` ``. Furthermore, one quick-fix named *Interface stubbing* has also been implemented. It implements missing object literal properties similar to how method stubs can be created for classes that do not correctly implement an interface. At the time of writing this document, the pull-request for *Convert lambda to function* has been reviewed and approved by Microsoft.

Management Summary

Roles

Graduates	Giovanni Heilmann, Arooran Thanabalasingam
Examiner	Thomas Corbat
Technical Advisor	Felix Morgner
Subject Area	Software-Engineering
Project Partner	Institute for Software

Motivation

Visual Studio Code is an increasingly popular text editor written in TypeScript. Language support and other features can be added via extensions. Both TypeScript and VS Code are open-source and maintained by Microsoft. The current TypeScript implementation has in-built language features like auto-complete, reference lookup, etc. Two such features are automated refactorings and quick-fixes. Refactorings are behavior-preserving code changes that make code clearer and more maintainable. Quick-fixes are code changes that change code behavior. They fix mistakes or extend existing behavior. Both features are invaluable not only for coding speed and efficiency, but also for creating maintainable code. Oftentimes, without refactoring, the code grows out of control and becomes unnecessarily complex. Only by refactoring can this complexity be reduced and the code be made more maintainable. Better maintainability reduces long-term maintenance costs and makes it easier and cheaper to change or add new functionality down the line. However, manual refactoring is time-consuming and error-prone. Automated refactorings make this process safe and quick.
Currently, VS Code lacks a lot of the essential refactorings for TypeScript like *inline variable* or *inline function*. Some of the very few refactorings it supports, are *Extract symbol* and *Move to new file*. The goal of this bachelor thesis is to contribute some of these refactorings and quick-fixes to the TypeScript repository

and thus improve the quality of life and development speed of all TypeScript developers.

Result

In total, four refactorings have been implemented: *Inline local*, *Inline function*, *Convert lambda to function*, and *Convert string concatenation to template literal*. Furthermore, one quick-fix named *Interface stubbing* has also been implemented. Judging from experience and according to ASERG at the Federal University of Minas Gerais, some of these features are among the most commonly used by developers, especially *Inline local* and *Inline function* which complement the already existing *Extract symbol*. [38] At the time of writing this document, the pull-request for *Convert lambda to function* has been reviewed and approved by Microsoft.

Outlook

Apart from the approved pull-request, the other ones have not yet been reviewed. This lies outside our responsibility and we have no influence over the review process. Once the pull-requests have been reviewed, any necessary changes will be made. This lies outside the scope of this bachelor thesis, since it has already concluded. Many common refactorings are yet to be implemented for TypeScript. To name a few: *Move method, Move class, Extract class, Extract interface, Pull up method, Push down method*. These and more can be the task for future projects.
Some issues still remain:
The implemented quick-fix, *Interface stubbing*, makes use of existing error messages. However, just before opening the pull-request, one of the error messages did not appear anymore. A related issue has been opened on Github regarding this matter. [32]
Apart from that, there is also the matter of the design flaw in the refactoring API, which prevents the language server to send diagnostic information to the client in case a refactoring fails. An appropriate issue has been openend on Github. [16]
Lastly, overloaded functions have been overlooked for the refactoring *Inline function*. This edge case needs to be implemented.

Contents

1. Introduction

This chapter serves as an introduction to this document. It explains essential terms and concepts used in the document.

1.1. VS Code

Visual Studio Code, also known as VS Code, is an open-source source code editor maintained by Microsoft. It runs on Windows, Linux and macOS. Apart from syntax highlighting, it offers support for embedded Git control, debugging and code actions like refactoring and quick-fixes among other features. In addition, it is customizable. Users can change keyboard shortcuts, user interface's theme and preferences. Furthermore, new features can be added through first and third party extensions. [37]
VS Code is written almost exclusively in TypeScript. [36] Similar to Atom, VS Code is built on Electron, which is a framework to run NodeJS applications in a desktop environment. [5]
VS Code was selected as the most popular developer environment tool in the Stack Overflow 2018 Developer Survey. [9]

1.2. TypeScript

TypeScript is a programming language that compiles to plain JavaScript. It is developed and maintained by Microsoft. It is a superset of JavaScript. This means that existing JavaScript code is also valid TypeScript code. That is why existing JavaScript libraries can easily be incorporated in a TypeScript project. Since TypeScript is compiled to plain JavaScript, TypeScript code can also be called from JavaScript code seamlessly. [35]
Types are optional in TypeScript and are provided through type annotation. If a type annotation is given, the type checking is done at compile time. The type annotations can explicitly be omitted in order to use the dynamic typing of JavaScript. In addition, the TypeScript compiler tries to infer the type of omitted types. Thanks to type inference, better readable code can be provided while still having type-safe code. [35]
The TypeScript compiler is written in TypeScript. [33]

1.3. Code Actions

In VS Code, a code action is an automated code modification performed by the editor. Apart from efficiency, the automated code actions are less error prone than the same modifications performed manually. There are two types of code actions: refactoring and quick-fix. They are explained in the following subsections. [11]

1.3.1. Refactoring

Refactoring is a technique that improves the quality and maintainability of a code base through restructuring of the code while preserving its observable behavior.
It is done in a series of tiny transformations without changing its external behavior. These isolated, standardized transformations are also called refactorings. Each transformation modifies only little, but an accumulation of these transformations can lead to a significant restructuring. Since the transformations are kept small, the refactoring is less error prone.
After each transformation, the system is kept fully working in order to reduce the risk that a system may get heavily damaged during the restructuring. This can be accomplished with an automated test framework with many and meaningful tests. [13]
In conclusion, *refactoring* can refer to the technique, a standardized code transformation, or a code action of an editor. In this document, refactoring usually refers to the code action. [11]

1.3.2. Quick-Fix

While programming, an editor is constantly analyzing the source code for potential faults. Wherever an issue occurs, the editor offers a list of proposals to fix or improve the code. Such a proposal, called quick-fix, is an immediate solution to the detected issue. Then, the user can choose the appropriate solution.
A quick-fix is not necessarily the optimal way of solving the detected issue for there may be deeper-lying faults. Most of the times, however, even such a suboptimal solution is sufficient to enable the application to run and start testing.

2. Analysis

This chapter describes the context of the project. It explains relevant interfaces and tools like the Language Server Protocol.

2.1. Architecture

In this section it will be shown how the VS Code editor can be extended.

2.1.1. Overview

The VS Code editor provides a common model to register and load extensions. Three kinds of extensions exist. The first is just called extension and it represents the base building block. The second is language server, where enhanced editing experiences can be provided by binding an external software. The language server will be explained in more detail in section 2.2. The third one is debugger, where an external debugger is wired up through a debug adapter. As shown in figure 2.1, the last two kinds have their own additional protocols. [10]
In order to maintain performance and compatibility, the extensions are run in a separate process and direct access to DOM is prevented. That means the VS Code editor can only be extended through the VS Code API. In addition, the extensions are only loaded in VS Code when they are needed. For example, a Java editing extension is only loaded when the user opens a Java file. [12]

2.1.2. Language support

Language support can be divided into basic support or advanced support based on implementation difficulty. Basic language support consists of syntax highlighting, snippets and smart bracket matching. Those can be provided just with configuration files. That means there is no need to write source code. Advanced language support has language features like code actions. There are two approaches to provide advanced language support. Using the direct implementation approach, the features are directly implemented as a VS Code extension. Secondly, when using the language server approach a VS Code extension wires up the services of language server with the VS Code API. [21]

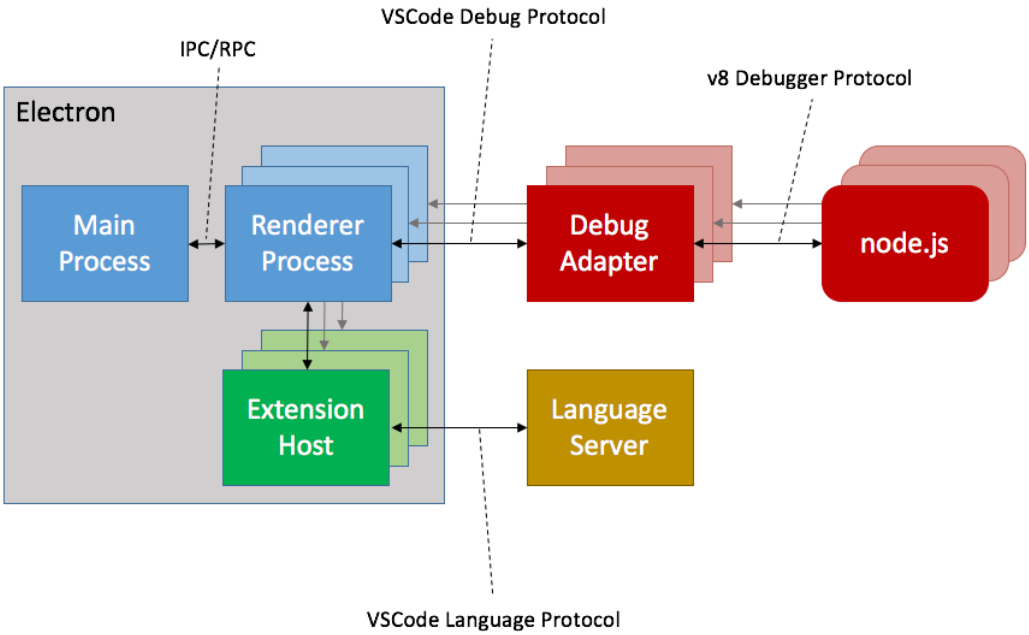

Figure 2.1.: Extensibility architecture Source: [10]

The extensions for the typescript language are directly maintained in code base of VS Code. [36] The basic language support is located in *extensions/typescript-basics/* and advanced language support in *extensions/typescript-language-features/*. The language server approach was chosen for the typescript language. The typescript language server itself is maintained in the code base of typescript. [33]

2.1.3. Patterns

The following patterns are used throughout the VS Code API.

Promises

Asynchronous operations are represented by promises in VS Code. Promise is abstracted with the type Thenable in order to be independent from any specific promise libraries. The common denominator for promises is the `then`-method. [12]

Cancellation tokens

Some operations can be started based on a volatile state. This state can change before the started operation is finished. For example, after the computation of auto-complete has been started, the user continues to type and thereby makes

result of the previous computation obsolete. Such behavior will pass a cancellation token. With this token it can be verified whether or not a cancellation has been triggered (`isCancellationRequested`). Furthermore, what should happen when a cancellation is fired can also be defined (`onCancellationRequested`). [12]

Disposables

The dispose pattern is used for resources which are obtained from VS Code. This includes event listening, commands, interacting with the UI, to name a few. For example, the `setStatusBarMessage` function returns a disposable. To remove the message from the status bar, the dispose method must be called on the disposable. [12]

Events

To subscribe to an event, a listener-function must be passed as an argument. Upon subscription an object implementing the `Dispose` interface will be returned. This disposable can later be used to unsubscribe. Events have the following naming convention formatted in regex [12]:
on[Will|Did]VerbNoun?

Strict null

Prior to typescript 2.0, null and undefined could be assigned to every type. Consequently, it was not possible to explicitly exclude them, and therefore it was not possible to detect any misuse. TypeScript 2.0 includes a new type checking mechanism called strict null checking. In strict null checking mode, `null` and `undefined` do not belong to the domain of every type as shown in the listing 2.1. They can only be assigned to themselves and `any`. [17]
To support strict null checking, VS Code uses `undefined` and `null`. [12] The codebase of typescript on the other hand only uses `undefined`. [6]

```
// Compiled with --strictNullChecks
let x: number;
let y: number | undefined;
let z: number | null | undefined;
x = 1;  // Ok
y = 1;  // Ok
z = 1;  // Ok
x = undefined;  // Error
y = undefined;  // Ok
z = undefined;  // Ok
x = null;  // Error
y = null;  // Error
z = null;  // Ok
```

Listing 2.1: Strict null checking examples Source: [17]

2.1.4. Communication between VS Code and TypeScript language server

Figure 2.2 shows the interaction between various actors in order to provide the requested refactoring service. There are five actors involved. The TypeScript extension represents the client extension for the TypeScript language server.
In this example, the user opens a TypeScript file, where a simple arrow function is contained. Then, the extension is launched, which automatically starts the TypeScript language server. The user then selects the arrow function and requests available refactorings. Once the server replies with the available ones, the user selects one of them, e.g. "Add braces to arrow function". Thus, the extension forwards the request to the language server, which applies the refactoring on the file and notifies the client of the success.

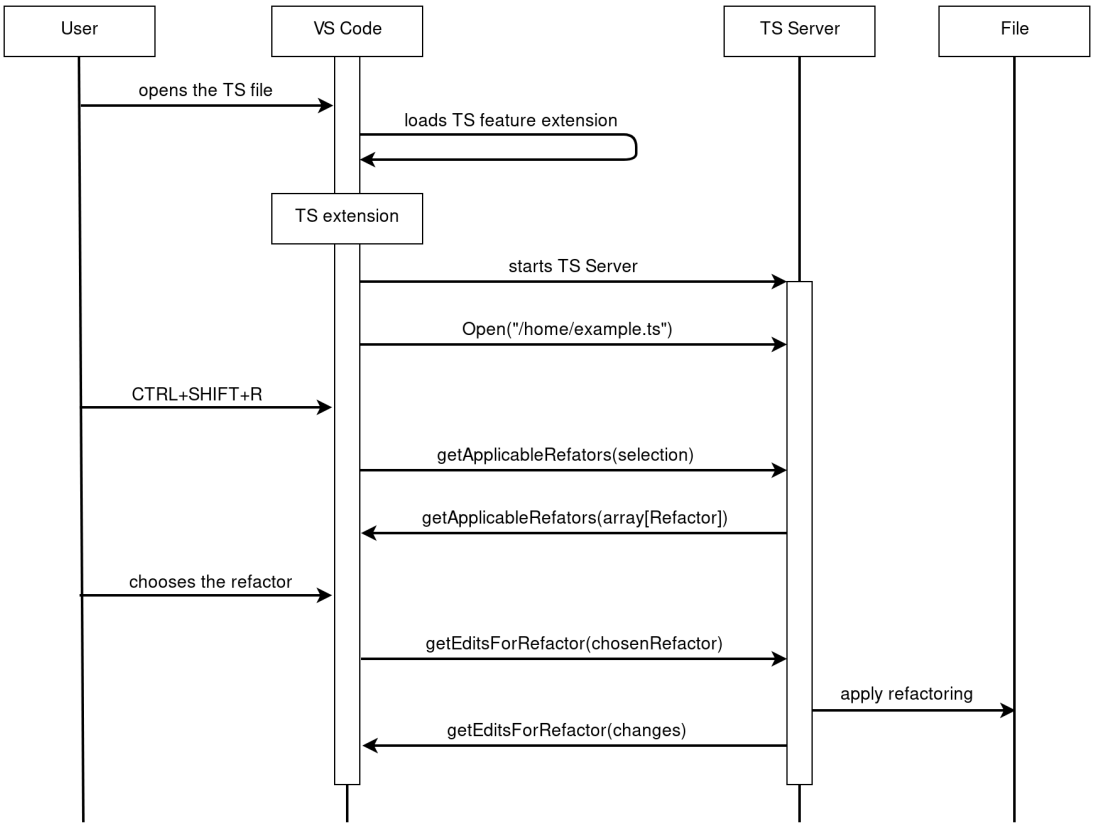

Figure 2.2.: Communication between VS Code and TS Server

2.2. Language Server Protocol (LSP)

This section provides an overview of the LSP and protocol details. Please note that the information provided in this section is not directly relevant to the project because the TypeScript language server does not implement it, and more importantly, implementing code actions can be implemented without knowledge of the language server protocol as described in section 2.3.2. Nonetheless, it provides context for better understanding of the environment and may be of use for future work. Visit the LSP overview [22] or the protocol specification [23] for further information.

2.2.1. What is the Language Server Protocol?

Implementing support for features like auto-complete, goto definition, or documentation on hover for a programming language is a significant effort. Traditionally,

this work had to be repeated for each development tool, as each provided different APIs for implementing the same features.
To clarify, let us look at an example. Multiple Java IDEs exist in today's market. Eclipse, IntelliJ, and Netbeans are just three of the many tools out there. And currently, each tool needs to implement their own language support, specifically syntax trees, syntax highlighting, code completion, refactorings, and so on. There is no common ground.

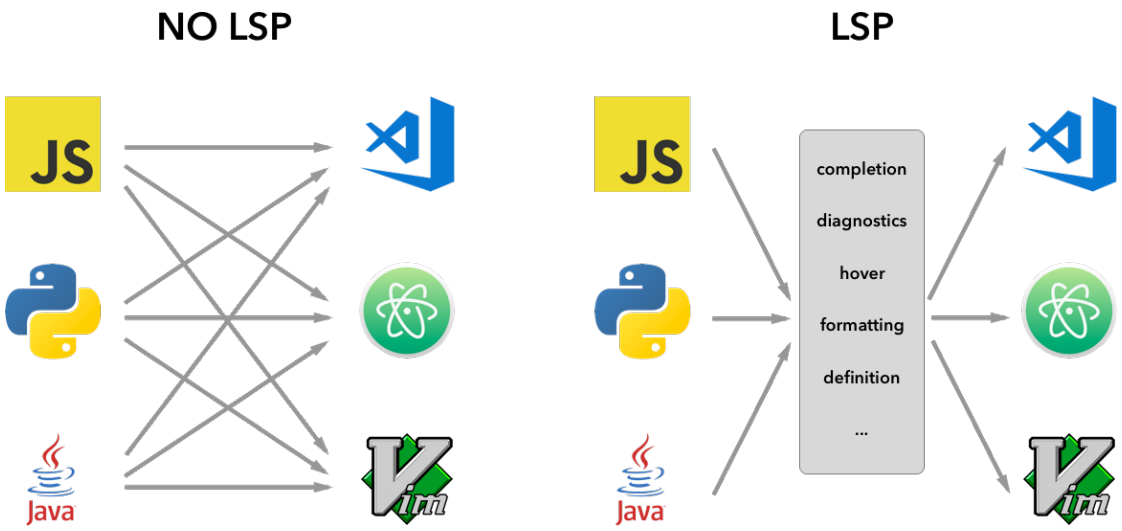

Figure 2.3.: A problem of complexity Source: [26]

However, all this work would ideally only be done once per language, since the functionality is the same. This is where the Language Server comes in. It gets rid of all the duplicated features by providing one implementation through a standardized interface. The LSP provides this interface.
This would mean for our Java example, that someone implements a Java Language Server with all the language smarts, and the clients (Eclipse, IDEA, etc.) can let that server do the heavy lifting. They only need to implement the front-end.

2.2.2. How does it work?

The language server runs in a separate process from the client. The client can communicate with the server using the LSP, which is based on JSON-RPC.
Figure 2.2.2 shows requests from a development tool to language servers and their responses. One can see that the request method consists of a data type (textDocument) and an action ([goto-]definition). This example shows that the

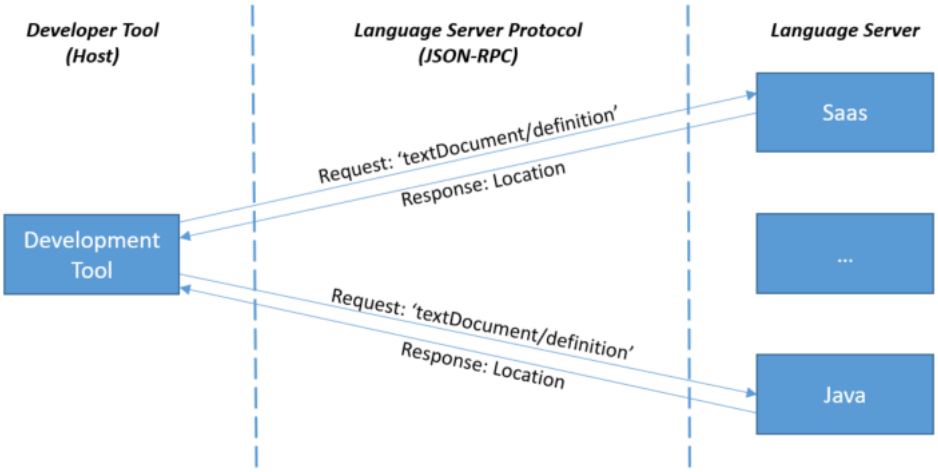

Figure 2.4.: Language Server Protocol communication Source: [10]

protocol uses document references and positions. This is universal to all text-based programming languages. It has the following benefit for designing the protocol:

> "It is much simpler to standardize a text document URI or a cursor position compared with standardizing an abstract syntax tree and compiler symbols across different programming languages." [22]

2.2.3. Specification

This section goes into more detail about the LSP. It explains the base protocol and the specifics.

Base protocol

As defined by JSON-RPC, Similarly to HTTP, every message consists of a header and a content part. The header contains a set of header fields, namely Content-Length and Content-Type. The latter describes the mime type of the content and can be omitted. In that case it defaults to `application/vscode-jsonrpc; charset=utf-8`. The content consists of a JSON object.
JSON-RPC defines three types of messages. Requests, responses and notifications. Requests sent by the client always require a response from the server, even if no result is expected. In that case the result must be `null`. Notifications are like events and do not require a response. Both clients and servers may send notifications.

```
Content-Length: ...

{
  "jsonrpc": "2.0",
  "id" : 1,
  "method": "textDocument/definition",
  "params": {
    "textDocument": {
      "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/use.cpp"
    },
    "position": {
      "line": 3,
      "character": 12
    }
  }
}
```

Listing 2.2: LSP request Source: [10]

Notifications and requests starting with `$/` can be ignored if the command is unknown, e.g. `$/cancelRequest`.

LSP specifics

Below, the most important points are summarized. For more detailed information, please refer to the LSP specification [23].

- Microsoft offers an npm module to parse document URIs.
- The protocol does not support binary document types. Only string-based documents are supported.
- Line positions cannot be relative to the end of a line.
- Document changes use the TextDocumentEdit interface. It contains a reference to the document in question and an array of type `TextEdit[]`.
- Bundled TextEdits' ranges may not overlap, but inserts at the same location are valid.
- If the start and end location of a range are the same, it signifies an insert.

```
interface Range {
  start: Position;
  end: Position;
}

interface TextEdit {
  range: Range;
  newText: string; // replaces text in range
}

interface TextDocumentEdit {
  textDocument: VersionedTextDocumentIdentifier;
  edits: TextEdit[];
}
```

Listing 2.3: TextDocumentEdit interface Source: [10]

2.3. Tsserver

This section describes the standalone language server that TypeScript uses called tsserver. Its interface differs from the LSP significantly, but is also based on a JSON protocol. For further reading, refer to the tsserver documentation. [30]

2.3.1. Key differences between LSP and tsserver interface

Historically, tsserver existed before LSP. [15] That is why the tsserver protocol and the LSP do not share a common interface despite both being developed by Microsoft. The goal is to have the tsserver implement the LSP in the future according to GitHub issue #11274. [20] This section will explain the key differences.

Sample request

To illustrate the differences of the JSON structures more closely as they appear on the wire, let us look at an example. Listings 2.4 and 2.5 show one request each for the tsserver protocol and LSP respectively. They both request possible refactorings for a given selection. As can be seen, the LSP interface is broken up into reusable blocks - textDocument, range, context -, whereas the tsserver interface has a flatter structure. Furthermore, the LSP interface is more flexible, since it can send additional diagnostic information and filters for refactorings, code fixes or both.

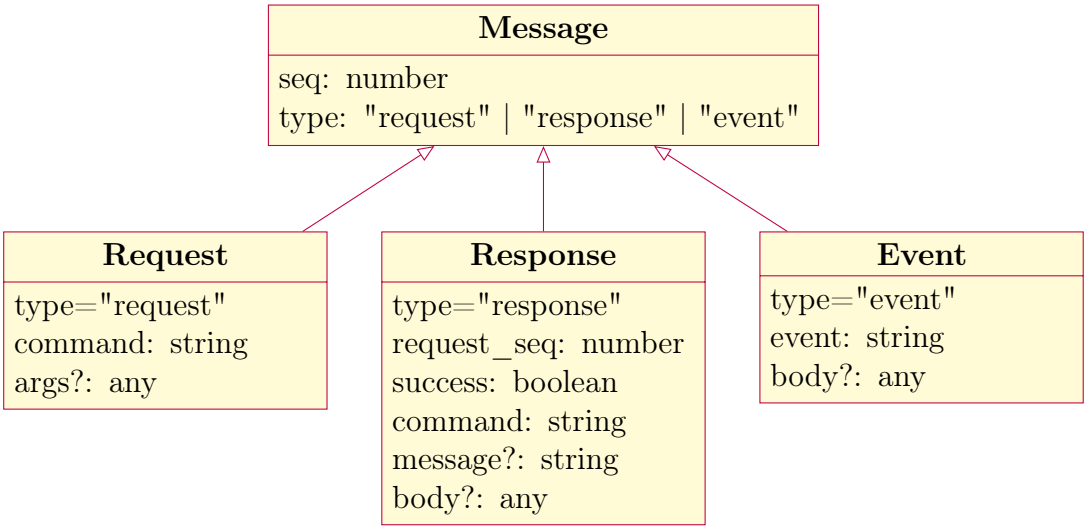

Figure 2.5.: tsserver protocol

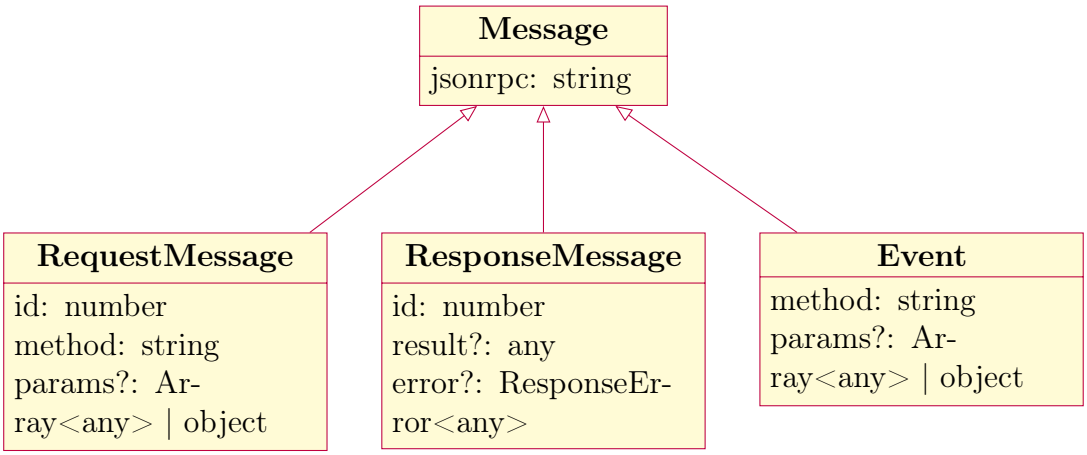

Figure 2.6.: Language Server Protocol

```
{
  "seq": 1,
  "type": "request",
  "command": "getApplicableRefactors",
  "args": {
    "file": "c:/path/to/file",
    "projectFileName": "..." // project name (optional)
    "startLine": 6,
    "startOffset": 24,
    "endLine": 9,
    "endOffset":42
  }
}
```

Listing 2.4: tsserver JSON: GetApplicableRefactors request

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/codeAction",
  "params": {
    "textDocument": {
      "uri": "file:///c:/path/to/file"
    },
    "range": {
      start: { line: 6, character: 24 },
      end : { line 9, character : 42 }
    },
    "context": {
      "diagnostics": {...} // diagnostic information relevant to context
      "only": {"refactor", ...} // code action type filter array
    }
  }
}
```

Listing 2.5: LSP JSON: Code action request

2.3.2. Implementing code actions

To implement code actions, it is not necessary to work on the tsserver interface. There is no work needed on the VS-Code side. Interfaces exist for both refactorings and quick-fixes, so developers can contribute without any knowledge of the tsserver protocol.
To make code changes, the AST needs to be transformed. Basic information on that is found in section 2.4.2.
The actual code changes are tracked by the `ChangeTracker` in the namespace `ts.textChanges` as seen in listing 2.6. The resulting `FileTextChanges` are then returned

to VS Code.
The lambda parameter in `with` can have one or more text changes depending on the code action.

```
1 const edits = textChanges.ChangeTracker.with(
2   context, t => t.replaceNode(file, oldNode, newNode)
3 );
```

Listing 2.6: Change tracker example

Refactoring API

As shown in figure 2.7, the interface for refactoring only requires the following two methods:

- **getAvailableActions**

  Using the information contained in `RefactorContext`, applicability of a given refactoring is verified. If refactoring actions are available, the possible actions are returned as `ApplicableRefactorInfo`. If no refactoring actions are applicable, then an empty array is returned.

- **getEditsForAction**

  This method computes the AST manipulations based on the `RefactorContext` information. On success, the AST manipulations are returned as `RefactorEditInfo`. Otherwise, it returns `undefined`.

Thanks to this decoupling of refactoring, the corresponding UI elements has only to be wired up once with the refactoring interface. Since the UI elements has already been wired up for VS Code, there is no need to implement anything on VS Code's side in order to provide new refactorings.
The refactoring descriptions and action descriptions are externalized in *src/compiler/diagnosticMessages.json* so that messages displayed to user can be localized at a later time.

Quick-Fix API

As shown in figure 2.8, the interface for quick-fix can have the following 4 members:

- **errorCodes**

  This property specifies all possible error codes, upon which the quick-fix's `getCodeActions` can be invoked. These can be already supplied by the compiler. However, it may be necessary to create own error messages.

- **getCodeActions**

  This function computes the AST manipulations for fixing a single faulty scenario based on the `CodeFixContext` information. On failure, this function returns `undefined`. On success, it returns the AST manipulations as an array of `CodeFixAction`.

- **fixIds**

  This property is optional. A fix id is an unique name for a certain quick-fix's action. This property specifies all the fix ids, upon which the quick-fix's `getAllCodeAction` can be called.

- **getAllCodeActions**

  This function is also optional, but when the property `fixIds` is specified, then this function must be provided. This function computes the AST manipulations for fixing multiple faulty scenarios based on the `CodeFixAllContext` information. It returns the AST manipulations as `CombinedCodeActions`.

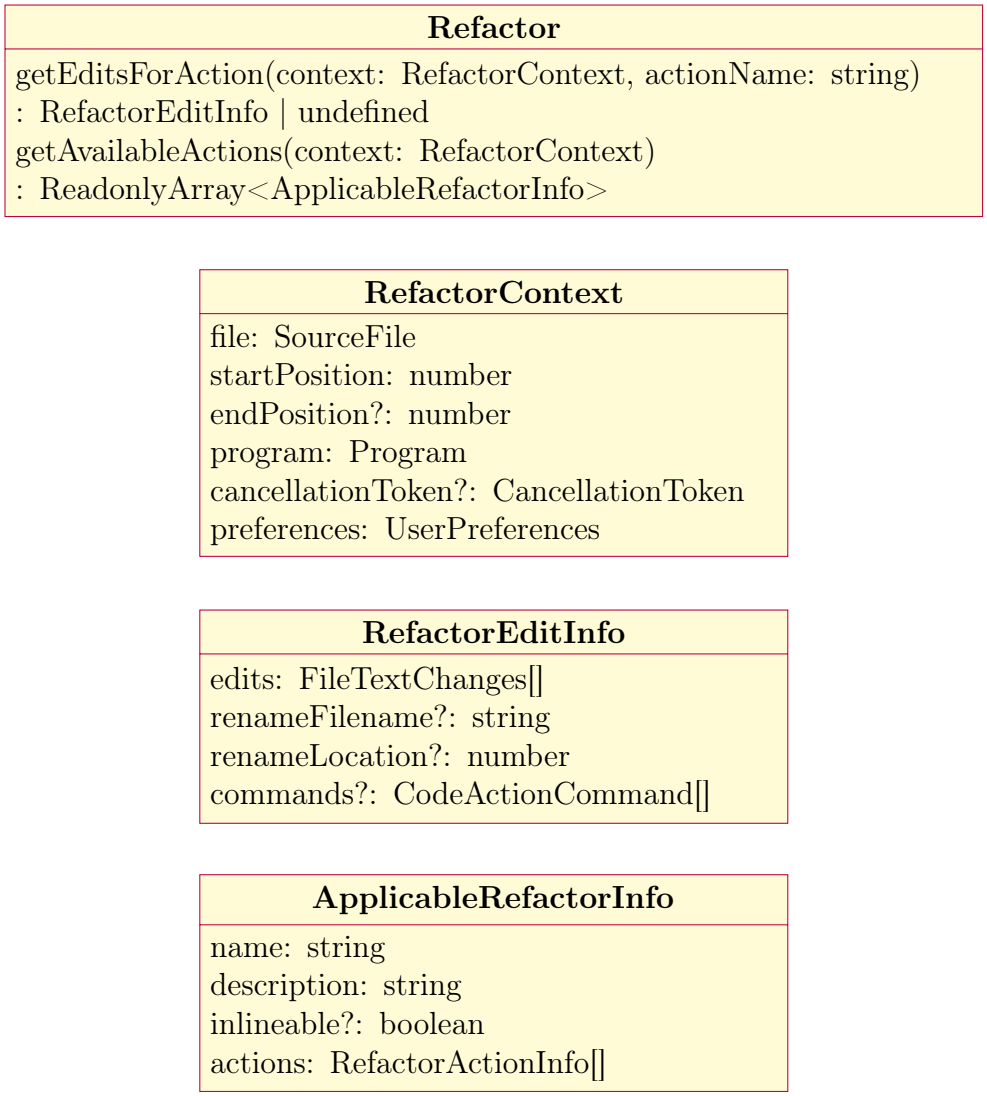

Figure 2.7.: Interface Refactor

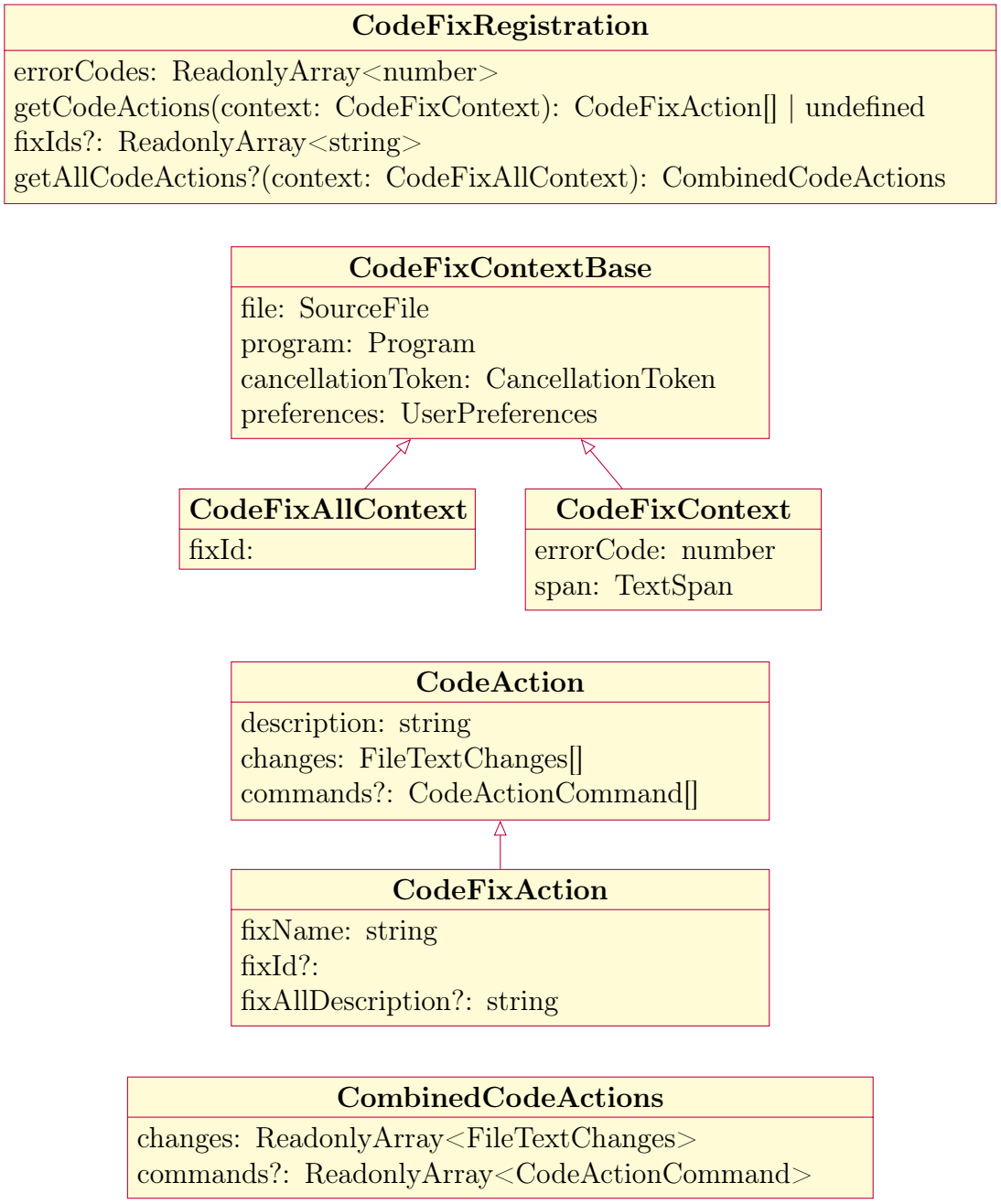

Figure 2.8.: Interface Quick-fix

2.4. TypeScript

This section describes the TypeScript language. It explains the compiler and its API. Information is taken from Basarat's book [4] and TypeScript's wiki [25].

2.4.1. Compiler overview

The TypeScript compiler reads the source file as a token stream, builds an AST, from which it generates a symbol table for semantics and type checking. The compilation process is described in the following section.
The scanner, controlled by the parser, reads the source file and generates a token stream. Tokens are a logical bundling of characters that allow the compiler to analyze the code. A token may be a number, a string literal, or an identifier like a type or variable name. There are more kinds of tokens.
The parser creates the AST from the token stream of a given file. Its type is `SourceFile` and it consists of a tree of `Node`s and additional information like the file name.
The binder generates `Symbol`s and binds them to nodes in the AST. Multiple `Node`s may be linked to the same `Symbol`. E.g., the code sample `int x = 41; x++;` contains the `Node` x twice, but since they are the same entity, they both share the same `Symbol`.
At this point, symbols are only unified file-wide. To get a global view of all symbols, a `Program` is created. It contains a list of `SourceFile`s and `CompilerOptions`. From the `Program`, a `TypeChecker` can be created. It is responsible for relationships between `Symbol`s from different files and their types. To do that, it first builds a global symbol table, merging equal `Symbol`s.
The `TypeChecker` is lazy. That is to say, it computes type information only on request. Irrelevant information is ignored.
Lastly, the `Emitter`, also created from the `Program`, generates the `.js` or `.d.ts`-files.

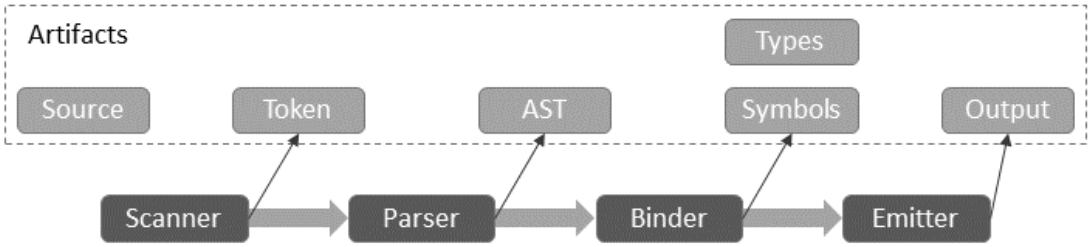

Figure 2.9.: Compiler Overview

2.4.2. Compiler API

The compiler provides an API for AST traversal and manipulation found in `src/compiler/factory.ts`.
It contains functions like `createSourceFile` or `updateFunctionDeclaration`. These are used to manipulate nodes instead of doing it by hand. These factory-methods should not be circumvented to edit AST nodes.

AST manipulation

The following example shows how to manipulate the AST. The code aims to replace the body of a selected function. Internally, the whole function is replaced as seen in figure 2.10.

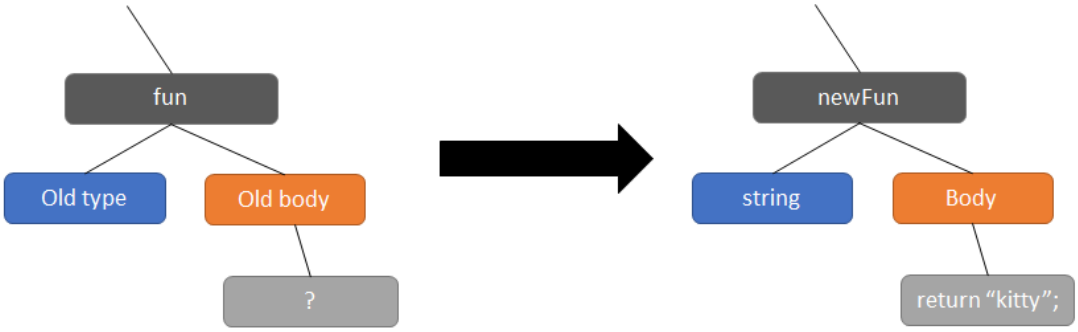

Figure 2.10.: AST manipulation

```
1  function getEditsForAction(
2      context: RefactorContext,
3      actionName: string): RefactorEditInfo | undefined {
4    const { file, startPosition } = context;
5    const info = isApplicable(file, startPosition);
6    if (!info) return undefined;
7
8    const {fun} = info;
9    const secretStr = createStringLiteral("kitty");
10   const statements = [createReturn(secretStr)];
11   const body = createBlock(statements, /*multiline*/ true);
12   const type = createKeywordTypeNode(SyntaxKind.StringKeyword);
13   const newFun = createFunctionDeclaration(
14     fun.decorators,
15     fun.modifiers,
16     fun.asteriskToken,
17     fun.name,
18     fun.typeParameters,
19     type,
20     body
21   );
22
23   const edits = textChanges.ChangeTracker.with(
24     context, t => t.replaceNode(file, fun, newFun));
25   return { edits };
26 }
```

Listing 2.7: Replace node example

First, a replacement node needs to be created with the API, as seen on lines 9 through 13. The node is a function with a single return statement. The method `createReturn` creates a return statement with the expression passed as an argument. The resulting statement is then passed to `createBlock`, which creates a statement block node. Furthermore, the function's type is changed accordingly.
After that, desired changes are tracked with the `ChangeTracker` on lines 15 & 16. The with-function requires a callback with the change in question. In this case, the original function node is replaced with `newFun`. Other actions include `insertNodeAfter` and `delete`.

Type query

As discussed in subsection 2.4.1, the `TypeChecker` is responsible for type resolution. The example code in listing 2.8 shows its usage.

```
1  const {scope, symbol, node} = getInfo();
2  const isJS = isInJSFile(scope);
3  let typeNode: TypeNode | undefined;
4  if (!isJS) {
5    const checker = context.program.getTypeChecker();
6    const type = checker.getBaseTypeOfLiteralType(
7      checker.getTypeOfSymbolAtLocation(symbol, node)
8    );
9    typeNode = checker.typeToTypeNode(
10     type,
11     scope,
12     NodeBuilderFlags.NoTruncation
13   );
14 }
15 node.type = typeNode;
```

Listing 2.8: Type query

The function `getTypeOfSymbolAtLocation` on line 7 may return a very constrained type with undesired annotations. Thus, `getBaseTypeOfLiteralType` is used to generalize the type. From this type, a `TypeNode` can be generated to be assigned to an AST-node.

2.5. Quick-Fix: Interface stubbing for object literal

This quick-fix is available, when an object literal implements an interface but the required members for interface are partially or fully missing. It creates missing members along with corresponding default values.
First, the object literal and its interface are analyzed to find the commonalities. Subsequently, intersection types, union types and type checking are also taken into account in order to gain deeper insights. Finally, each case is examined separately.

2.5.1. Interface

An interface helps defining contracts within code. It can contain properties, methods, indexers and function signatures. But, function signature is not relevant for objects.
The indexer describes, which type can be used for indexing into the object and it also defines the return type of indexing. Since properties of an object can be accessed through indexer in JavaScript, there is no need to explicitly provide an indexing function as demonstrated in listing 2.9. [19]

```
interface Example {
    prop: string;                       // Property
    optProp?: string;                   // Optional property
    readonly readProp: number;          // Readonly property
    method(): void;                     // Method

    [propName: string]: any;            // Indexer
}

let obj: Example = {
    prop: "stringo",
    readProp: 42,
    method(){}
};
obj["optProp"] = "for accessing elements";
```

Listing 2.9: Interface with properties and indexer

As shown in listing 2.10, a function signature defines the parameter types and the return type of a function. It is only meant to be used in combination with function expressions and not with objects. [19]

```
interface Finder {
 (array: number[], element: number): number  // Function signature
}
let positionFinder: Finder;
positionFinder = function (array: number[], element: number) {
  return -1;
}
```

Listing 2.10: Interface with function signature

An interface can inherit from other interfaces as well as from classes. Inheriting from a class is the same behavior as inheriting from an interface as demonstrated in listing 2.11. In case the inherited class contains a private property, the interface becomes useless because it cannot have private property. [19]

```
interface Drive { speed: number }
class Vehicle { seats: number = 0 }

interface Car extends Drive, Vehicle {}
let myCar: Car = { speed: 120, seats: 2 };

class Tank { private gun: string = "75mm" }
interface SpeedyTank extends Tank {}
let uselessTank: SpeedyTank = { gun: "25mm" };
// Property 'gun' is private in type 'SpeedyTank'
// but not in type '{ gun: string; }'.
```

Listing 2.11: Interface inheritance

2.5.2. Object literal

Apart from using a constructor `new Object()` or the static method `Object.create()`, JavaScript allows the use of the object literal notation `{ }` to create a new object. An object literal contains a list of properties. Each property has a name and a value as demonstrated in listing 2.12.
Since ECMAScript 6, it is possible to declare properties by writing only its name if such a exists in that scope as shown on line 8. In addition, the property's name can be computed dynamically.
Apart from value properties, object literals can also contain methods, get/set accessors and spread properties. With the spread operator ..., properties of an object can be copied into a new object as demonstrated on line 18. When using the same name for a property, the value will be overwritten as shown on line 18. [28]

```
1
2  let foo = "Daisy", bar = false;
3
4  const newObj = {
5    greeting: "Hi there",    // Property
6    _magicNo: 42,            // Property
7    foo: foo,                // Property
8    bar,                     // Shorthand property
9    ["computed"+33]: true    // computed propert name
10   method(){return "nada"}, // Method
11   get magicNo(){ return this._magicNo }, // Get accessor
12   set magicNo(no){ this._magicNo = no }  // Set accessor
13 }
14
15 let obj1 = { foo: 'duck', x: 42 };
16 let obj2 = { foo: 'bear', y: 13 };
17
18 const mergedObj = { ...obj1, ...obj2 };   // spread property
19 // Object { foo: "bear", x: 42, y: 13 }
```

Listing 2.12: Object literal example

JSON vs object literal

The object literal is not the same thing as the JavaScript Object Notation (JSON). Although they both look very similar, there are differences. In JSON, a property's name must be double-quoted and it cannot be declared in a shorthand. JSON cannot have function declarations like methods or accessors. Furthermore, JSON allows only a handful of types. [28]

2.5.3. Intersection types

An intersection type combines multiple types into one type. As shown in listing 2.13, an object of intersection type must have all members of all types from the intersection and therefore, the `eagle` is both `Bird` **and** `Predator`. The ampersand is used to combine types. Intersection types are commonly used for mixins. [1]

```
1  interface Bird {
2      WingLength: number;
3      Height: number;
4  }
5  interface Predator {
6    Prey: string;
7  }
8
9  const eagle: Bird & Predator = {
10   WingLength: 2,
11   Height: 0.4,
12   Prey: "Rabbit"
13 };
```

Listing 2.13: Intersection type example

2.5.4. Union types

A union type defines a type which can be one of the specified types. In other words, `pet` in listing 2.14 can either be `Eagle` **or** `Magpie`. If a variable has a union type, then it is only possible to access members that are common to all types in the union as demonstrated on line 16. As long as the real type is not clear, it is only possible to work with intersection of `Eagle` or `Magpie` as shown on line 16. The vertical bar is used to separate each type as shown. [1]

```
1  function parseInt(value: string | number) { ... }
2  parseInt("foobar4 rocks");    // Okay
3  parseInt(42.233);             // Okay
4  parseInt(false);               // Compile-Time Error
5
6  interface Eagle {
7      fly(): void;
8      hunt(): void;
9  }
10 interface Magpie {
11     fly(): void;
12     steal(): void;
13 }
14 function getBird(): Eagle | Magpie { ... }
15
16 let pet = getBird();
17 pet.fly();       // Okay
18 pet.steal();     // Error
```

Listing 2.14: Union type example

2.5.5. Type checking

```
1  interface Bulk {
2    width: number;
3    height?: number;
4  }
5
6  function applySize(dimension: Bulk){}
7
8  let threeDim = { width: 42, height: 52, depth: 33 };
9  applySize(threeDim);                                          // Okay
10
11 applySize({ width: 42, height: 52, depth: 33 });              // Error
12 let bulk3d: Bulk = { width: 42, height: 52, depth: 33 }; // Error
13 // Object literal may only specify known properties,
14 // and 'depth' does not exist in type 'Bulk'.
15
16 applySize({ width: 42, hight: 52 });                         // Error
17 applySize({ width: 42, height: 52, depth: 33 } as Bulk); // Okay
```

Listing 2.15: Excess property checking

The error on line 11 of listing 2.15 might be confusing at first glance, since the object literal fulfills the interface's requirements like the `threeDim` object. Nevertheless, an error is reported.
The reason is that TypeScript treats object literals regarding type checking in

a special way. If an object literal is assigned to a variable or it is passed as an argument, it undergoes *excess property checking*. That means if an object literal has any properties that the target type does not possess, an error will be reported. This general attitude is taken in order to avoid situations like on line 16. Since the property `height` is optional, the misspelling of height would fail silently in JavaScript. Thanks to excess property checking, such mistakes are detected.
If code authors are confident about their code, they can simply circumvent this check with a type assertion as shown on the line 17. [19]

2.5.6. Conclusion of considerations

After taking interface and object literal into account, the common denominator of member kinds are property and method. Since optional properties are allowed to be omitted, they will be left out.
For the following reasons, accessors, shorthand/spread properties, function signatures and indexers will be ignored for further procedures: Get/set accessors cannot be enforced through an interface. An implementation for indexers cannot be provided because they are automatically generated. Shorthand and spread property are convenient features, but for interface stubbing they are not useful.
If an interface inherits from a class with private member, then this interface is already broken. Since the checker reports this error accordingly and only humans can fix such an error, no further steps are taken to solve this problem.

2.5.7. Cases

Since an object literal can either be assigned to a variable or passed as function argument, the following existing error messages are considered suitable to invoke this quick-fix:

- `Type_0_is_not_assignable_to_type_1`
- `Argument_of_type_0_is_not_assignable_to_parameter_of_type_1`

As function argument

In the following situation, a function parameter requires a certain interface, but the argument is an empty object literal. As demonstrated in listing 2.16, the empty object literal must be completed with all the properties of the required interface.

```
interface Foo {
  field: string
}
function example(arg1: string, arg2: Foo, arg3: number) { ... }

// before
example("arg1", {}, 0);

// after
example("arg1", {
  field: ""
}, 0);
```

Listing 2.16: As function argument

Basic types

This is the other base case where a variable requires a certain interface and an empty object literal is assigned. For such simple types, a property with the default values will be created as shown in listing 2.17.

```
interface Foo {
    varchar: string,
    numero: number,
    boolo: boolean,
    array: number[],
    zero: null,
    every: any
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    varchar: "",
    numero: 0,
    boolo: false,
    array: [],
    zero: null
    every: "any",
};
```

Listing 2.17: Basic types

Objects

The constructor is called for class instances as well as for primitive wrapper objects, e.g. `Boolean` or `String`. The type `object`, written in lower case, represents any non-primitive type. The type `Object`, written in upper case, describes functionality that is common to all objects. [29] As demonstrated in listing 2.18, the default value for an enum is its first value.

```
class Building {}
enum Direction { Up, Down, Right, Left }

interface Foo {
    clazz: Building,
    bigObj: Object,
    str: String,
    nested: { a: { b: string}},
    smallObj: object,
    enumo: Direction
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    clazz: new Building(),
    bigObj: new Object(),
    str: new String(),
    nested: {
        a: {
            b: ""
        }
    },
    smallObj: new Object("anyObject"),
    enumo: Direction.Up
};
```

Listing 2.18: Objects

Methods

Function stubs are always implemented with a throw expression so that the user will be notified in case of accidental use. There are two ways to declare a function in an interface: One is as a method and the other is as an arrow function as shown in listing 2.19.

```
interface Foo {
    methodo(a: string, b: number): boolean,
    lambda: (a: number, b: string) => void
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    methodo: (a: string, b: number): boolean => {
        throw new Error("Function not implemented.")
    },
    lambda: (a: number, b: string): void => {
        throw new Error("Function not implemented.")
    }
};
```

Listing 2.19: Methods

Intersection and union types

In case of an intersection type, default values must be provided for all members of all types. When a member's name is the same as that of another type, they must all have same type. As demonstrated in listing 2.20, a default value for the duplicated members should only be provided once. If the duplicated members have different types, then it is an error which can only be resolved by a human. In case of a union type, a default value must only be provided for one of the types.

```
interface Z { z: boolean; y: number }
interface X { x: string; y: number }

interface Foo {
    intersection: X & Z,
    union: string | number,
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    intersection: {
        x: "",
        y: 0,
        z: false
    },
    union: ""
};
```

Listing 2.20: Intersection and union types

Tuples

Tuples can contain almost any type, even an arrow function. For the sake of simplicity, not all types are mentioned in listing 2.21.

```
interface Foo {
    basic: [string, number, Boolean],
    fn: [() => string],
    nested: [number, [number, [string, boolean]]]
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    basic: ["", 0, new Boolean()],
    fn: [(): string => {
        throw new Error("Function not implemented");
    }],
    nested: [0, [0, ["", false]]]
};
```

Listing 2.21: Tuples

Type alias

The real types can be hidden behind a type alias. In such cases, the default values must still be provided. Again, not all types are mentioned in listing 2.22.

```
interface Z { z: boolean; }
interface X { x: string; }
interface A { o: Z; }
interface B { o: X; }

type Intersection = A & B;
type Union = A | B;
type ArrowFn = (a: string) => void;
type ObjLiteral = {a: number};

interface Foo {
    intersection: Intersection,
    union: Union,
    arrowFn: ArrowFn,
    obj: ObjLiteral
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    intersection: {
        o: {
            z: false,
            x: ""
        }
    },
    union: {
        o: {
            z: false
        }
    },
    arrowFn: (a: string): void => {
        throw new Error("Function not implemented.")
    },
    obj: {
        a: 0
    }
};
```

Listing 2.22: Type alias

Generics

Classes or functions can be generic. As shown in listing 2.23, the important thing is that type arguments for generic are copied correctly.

```
class Box<T> {}
interface Foo {
    clazz: Box<Building>,
    arrow: <T>(a: T) => boolean,
    fn<E>(a: E, b: E): void
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    clazz: new Box<Building>(),
    arrow: <T>(a: T): boolean => {
        throw new Error("Function not implemented.")
    },
    fn: <E>(a: E, b: E): void => {
        throw new Error("Function not implemented.")
    }
};
```

Listing 2.23: Generics

Inheritance

An interface may inherit from another interface or from a class. Accordingly, default values must be provided for the interface's members as well as for any inherited members as demonstrated in listing 2.24.

```
class Base1 {
    state: boolean
}

interface Base2{
    attribute: number
}

interface Foo extends Base1, Base2 {
    field: string
}

// before
const bar: Foo = {};

// after
const bar: Foo = {
    state: false,
    attribute: 0,
    field: ""
};
```

Listing 2.24: Inheritance

Partially existing and nullable members

An object literal can already contain some members of the target interface. Subsequently, only the default values for the missing members must be provided. If a property is optional, then it will be ignored as shown in listing 2.25.

```
interface Foo {
    field1: string,
    field2: boolean,
    field3: any,
    optField?: number
}

// before
const bar: Foo = {
    field3: "occupied",
    field1: "conquered"
};

// after
const bar: Foo = {
    field2: false,
    field3: "occupied",
    field1: "conquered"
};
```

Listing 2.25: Partially existing and nullable members

2.6. Refactoring: Lambda to function

This refactoring converts an arrow function, the Javascript-specific name for lambda function, to a function and vice-versa. Arrow function and lambda will be used interchangeably in this document. The considerations made are documented in this section. First, there is a set of general considerations that are universally valid. Then, each case is looked at separately.

2.6.1. General considerations

There are a few things that are equally important for all cases. These will be listed here.

Closure

Closure is a concept defined by JavaScript and thus also relevant for TypeScript. In short, any function can access symbols from outer scopes. This is also valid for lambdas. In the example listing 2.26, `foo` can access `bar1` and `bar2`, but not `bar3`, because it is in a different scope.

```
let bar1 = 6;
function func () {
  let bar2 = 9;
  let foo = bar1 * 10 + bar2 - bar3;
}

function funky () {
  let bar3 = 42;
}
```

Listing 2.26: Closure

In the case of this refactoring, it is not relevant, since functions behave the same as lambdas. Furthermore, there is no change in scope, i.e. no function is moved anywhere else. The refactoring is a strict transformation. The same goes for externally defined constants and global variables. They are visible before and after the operation.

Type considerations

The refactoring is a mere conversion. The return and parameter types stay the same.

Comments

When implementing the refactoring, comments must be preserved in the process. Since they usually are not part of the syntax tree, they must be handled with care. Listing 2.27 shows possible locations of comments to be considered.

```
const foo1 = (a) => /*c1*/ a+1; //c2
const foo2 = (a) => { /*c3*/ return a+1; }; //c4
```

Listing 2.27: Comments

A preliminary study showed that `c1` is not preserved by default, but `c3 & c4` are. Further discussions are necessary to determine whether `c1` is an actual use-case and needs to be preserved.

2.6.2. Case by case considerations

There are four cases to consider:

- To anonymous function
- To named function
- To arrow function

- Lambda expression to named function declaration

Each case has to be given consideration, which will be discussed below.

To anonymous function

This action converts a lambda expression to a function expression and vice-versa. Listing 2.28 shows an example for this refactoring:

```
1 function someFoo(foo) { foo(); }
2
3 someFoo(() => "beta");
4
5 // <->
6
7 someFoo(function () {
8     return "beta";
9 });
```

Listing 2.28: Lambda to anonymous function

One important consideration is that a lambda expression may have an expression or a statement block as a body, but a normal function only accepts a statement block. A conversion is therefore necessary.
The following consideration has been made post-development and is not reflected in the code: This action should be called *To function expression*, because function expressions can have a name as seen earlier. Thus, *To anonymous function* is a misnomer.

To named function

This action converts a lambda declaration as seen in listing 2.29 to a named function.

```
1 // before
2 const alpha = () => "beta";
3
4 // after
5 function alpha() {
6     return "beta";
7 }
```

Listing 2.29: Lambda to named function

There are certain edge cases to be considered as well. For example, when there are multiple variable declarations in the same statement as seen in listing 2.30. In this case, the function must be extracted to a new line and the other declarations must be left intact.

```
// before
let a, b = () => "DragonBall";

// after
let a;
function b() {
    return "DragonBall";
}
```

Listing 2.30: Multiple declarations

Another thing to consider are modifiers like `export`. Modifiers must be kept when converting.

```
// before
export let a = () => "DragonBall";

// after
export function a() {
    return "DragonBall";
}
```

Listing 2.31: Modifiers

To arrow function

There are two cases this action could be applied:

- Function expression to lambda expression
- Function declaration to lambda variable declaration

Listing 2.32 shows the first case:

```
// before
function someFoo(foo) { foo(); }

someFoo(function () {
    return "beta";
});

// after
function someFoo(foo) { foo(); }

someFoo(() => "beta");
```

Listing 2.32: Function expression to arrow function

A conversion to an expression body is done if the function body contains only one statement and that statement is a return statement.

Furthermore, a function expression can have a name, but it is not required. Lambdas, however, cannot have a name. Thus, a function expression must not be converted to a lambda if it has a name that is referenced. As it is illustrated in listing 2.33: This was brought to our attention by Andy Hanson on Github. [14]

```
1 function someFoo(foo) { foo(); }
2
3 someFoo(function fac(n) {
4     return n > 1 ? n * fac(n-1) : 1;
5 });
```

Listing 2.33: Function expression with referenced name

The second case converts a named function declaration to a lambda declaration. This case must not be supported because of hoisting – or rather the lack thereof. JavaScript implements hoisting, which enables function declarations and `var` variables to be visible before they are declared. However, `const` and `let` do not share this behavior. As shown in listing 2.34, if the function `hoisted` was converted into the lambda declaration `trap`, then it would change the visibility.

```
1 trap(); // not visible
2 hoisted(); // visible
3
4 const trap = () => "It's a tarp!";
5 function hoisted() {
6     return "Yo, ho, and a bottle of rum";
7 }
```

Listing 2.34: Hoisting

Lambda expression to named function declaration

When researching how the refactoring should work and what it should support, Webstorm was used as a reference. With Webstorm, it is possible to convert a lambda expression to a named function using the code action "Convert to named function" as can be seen in listing 2.35. However, this is rather an extraction, not strictly a conversion, as pointed out by Kingwl in the issue's thread [7]. Another reason, why this should not be implemented, is that the refactoring *Extract symbol* already supports it.

```
1 // before
2 applyFn(c => "gamma");
3
4 // after
5 applyFn(gammaFn);
6
7 function gammaFn(c) {
8     return "gamma";
9 }
```

Listing 2.35: Lambda expression extracted

2.7. Refactoring: Inline variable

This refactoring replaces one or all usages of a local variable with its definition and deletes it if there is no usage left. This section talks about considerations made for this refactoring.

2.7.1. Simple case

This is the most basic case with one declaration and one usage as shown in listing 2.36. In this case, `someexpression` should replace `variable` as the function argument. More generally, any variable usage should be replaced with its value.

```
1 // before
2 const variable = someexpression;
3 foo(variable);
4
5 // after
6 foo(someexpression);
```

Listing 2.36: Inline variable - simple case

One must be able to invoke the refactoring from the declaration or usage. More on that in the following case.

2.7.2. More than one usage

If the same variable is referenced more than once, the user can inline all usages or just one. When inlining only one usage, the variable declaration must not be deleted.

```
1  // before
2  const variable = expr;
3  foo(variable);
4  const copycat = variable + otherexpr;
5
6  // after - ok
7  foo(expr);
8  const copycat = expr + otherexpr;
9
10 // after - problematic
11 foo(variable); // variable undefined
12 const copycat = expr + otherexpr;
```

Listing 2.37: Inline variable - multiple usages

One must be able to invoke the refactoring from the declaration or any usage. Two options should be available depending on context:

- Inline all
  - Inline all usages
  - Delete declaration
  - Invokable from declaration or usage
- Inline here
  - Inline one usage
  - Do not delete declaration
  - Invokable only from usage

2.7.3. Member access

There are cases when inlining a variable produces garbage code. For example, in listing 2.38 the inlined variable is an new object declaration. Each replacement creates a new object, which does not preserve behavior. The user may invoke the refactoring at his own risk, this is not to be prevented.

```
1  // before
2  const variable = new Class();
3  variable.member();
4  foo(variable);
5
6  // after
7  (new Class()).member();
8  foo(new Class());
```

Listing 2.38: Inline variable - nonsense refactoring

2.7.4. Assigned more than once

In the case that the variable is assigned more than once like in listing 2.39, it is unclear which value should be used. Therefore, one should not be able to invoke the refactoring at all.

```
1 let variable = someexpression;
2 variable = otherexpression; // culprit
3 foo(variable);
```

Listing 2.39: Inline variable - assigned twice.

2.7.5. Declaration not initialized

In case the variable is not initialized at declaration, the refactoring must not be invoked, since there is no value given. One consideration is to allow it if it is assigned later on, but this becomes ambiguous regarding the previous case.

2.7.6. Special keywords

Refactoring behavior may change depending on the keyword used. There are only two keywords that can be used in the context of variable declaration: `export` and `default`. The latter can furthermore only be used in conjunction with the former. Effectively, only `export` needs to be considered. This keyword signifies that the tagged variable is exported from the module. In this case, inlining deletes a public variable, which could be problematic. Thus, one must not be able to invoke the refactoring.

2.7.7. Operator precedence

Operator precedence comes into play if both of the following conditions are met:

- the inlined variable is a binary expression and
- an operand of a binary or unary expression

In case these two conditions are met, it may be necessary to surround the inlined expression with parentheses. This depends on the operator precedence given by the TypeScript language. Generally, if the inlined binary expression's operator precedence is lower than the outer one, it needs to be parenthesized. If it is higher, then not. If the outer and inner precedences are the same, further considerations are necessary. This is true even if the no binary expressions are involved, for example with unary expressions. If the target node's parent is indeed a unary

expression, parentheses are only necessary, if the parent's operator precedence is higher. Both other cases can safely be ignored.

- Left-associated operator: parenthesize if right operand (e.g. ==, ||)
- Right-associated operator: parenthesize if left operand (i.e. **)
- Commutative operator: no parentheses necessary (i.e. +, *)

```
1 // before
2 const sum = 2 + 4;
3 const prod = 9 * sum;
4
5 // faulty after
6 const prod = 9 * 2 + 4;
7
8 // correct after
9 const prod = 9 * (2 + 4);
```

Listing 2.40: Inline variable - operator precedence

2.7.8. Unary operators

The unary operators ++ and -- can only be used with variables. Inlining that variable may break the behavior but that is not caught by the refactoring. This case is ignored.

```
1 // before
2 let sum = 4 + 2;
3 const prod = 9 * --sum;
4
5 // after
6 const prod = 9 * --(4 + 2);
```

Listing 2.41: Inlining

2.8. Refactoring: Inline function

This refactoring replaces one or all function calls with the body of its referenced function.

2.8.1. General

All statements in the declaring function except for the return statement are inserted before the call statement. If the function returns a value, the call is replaced with the function's return expression.

```
1  // before
2  function foo() {
3    const sod = 42;
4    return sod;
5  }
6  function bar() { const meaningOfLife = foo(); }
7
8  // after
9  function bar() {
10   const sod = 42;
11   const meaningOfLife = sod;
12 }
```

Listing 2.42: Inline function - base case

The refactoring has two actions:

- Inline all:
  - Inline all occurences
  - Delete function declaration
- Inline here:
  - Inline selected occurence
  - Do **not** delete function delcaration

Inline all is available on both function declaration and calls. *Inline here* is available on function calls.

2.8.2. Special keywords

Similar to *2.7 Refactoring: Inline variable*, the refactoring may change the behavior depending on keywords. In addition to the two already known keywords `export` and `default`, the following candidates exist:

- `async`: signifies asynchronous function
- `public`: changes class method visibility
- `private`: changes class method visibility
- `protected`: changes class method visibility
- `static`: method can be called on class and not only on instances

They keyword `static` is the only one which can potentially change behavior significantly. But it should still be allowed if the user requires it for some reason.

2.8.3. Operator precedence

If the return expression of a function is not simple, it may be required to parenthesize it. Refer to section *2.7 Refactoring: Inline variable* for specifics, since the behavior is the same.

2.8.4. Closure

If a function contains closures, the referenced symbols may not necessarily be defined in the target location. If that is the case, then the refactoring is not permissible.

2.8.5. Parameters

If the function has parameters, a new variable declaration must be created for each one. The declarations are initialized with the arguments of the function call.

```
1 // before
2 function square(arg: number) { return arg*arg; }
3 function bar() { return square(2); }
4
5 // after
6 function bar() {
7     const arg = 2;
8     return arg*arg;
9 }
```

Listing 2.43: Inline function - parameters

Destructured parameters

Destructured parameters must be handled as well. Two kinds of destructuring exist: object destructuring and array destructuring. Only array destructuring is shown below:

```
// before
function foo({arg0, arg1}) {
    return arg0 * arg1;
}
function bar() {
    return foo({ arg0: 21, arg1: 2 });
}

// after
function bar() {
    const { arg0, arg1 } =  { arg0: 21, arg1: 2 };
    return arg0 * arg1;
}
```

Listing 2.44: Inline function - object destructuring

2.8.6. Multiple return statements

Functions with multiple return statements can be inlined if they are not of `void` type. First, a new variable declaration is created. Then, all return statements are replaced with assignments to the variable. Lastly, the function call is replaced with this variable.

```
// before
function foo() {
    const x = 2;
    if (x < 0) return 42;
    else return 69;
}
function bar(arg: number) {
    const someValue = foo();
    return arg * someValue;
}

// after
function bar(arg: number) {
    const x = 2;
    let expr;
    if (x < 0) expr = 42;
    else expr = 69;
    const someValue = expr;
    return arg * someValue;
}
```

Listing 2.45: Inline function - multiple return statements

Functions of `void` type cannot be inlined as is, because of how the return statement changes the program flow. The return after `doOneThing` in listing 2.46 breaks the

flow. `doFinally` is not executed if the condition is true. Thus, `void` type functions are not inlined. Other IDEs like Webstorm do not inline functions with multiple return statements at all.

```
function foo() {
  if (condition) {
    doOneThing();
    return;
  }
  else {
    doOtherThing();
  }
  doFinally();  // is only executed if condition is false
}
```

Listing 2.46: Inline function - multiple return statements

2.8.7. Name conflict

Local names may conflict with those in the target location. Thus, it is imperative to resolve this name conflict. Each conflicting local name needs to be replaced with a unique name. This is done for any parameter and other local declarations like variables and functions.

```
// before
function square(arg: number) { return arg*arg; }

function bar(arg: number) {
  return arg * square(2);
}

// after
function bar(arg: number) {
  const arg_1 = 2;
  return arg * arg_1 * arg_1;
}
```

Listing 2.47: Inline function - name conflict

2.8.8. Throws

In case the function does not exit in a controlled way, e.g. via a `throw` statement, inlining that function changes the behavior because more code will be skipped if a `throw` occurs. The user should be aware of this problem. Thus, this case will be ignored.

2.8.9. Overloading

Overloading in TypeScript is allowed as long as there are no duplicate function definitions. This can be used to make APIs more understandable instead of having to look at a long list of optional parameters.

```
1  // not allowed
2  function square() { return undefined; }
3  function square(n: number) { return n * n; }
4
5  // before
6  function square(): number;
7  function square(n: number): number;
8  function square(n?: number) { return n * n; }
9  const result = square(2);
10
11 // after
12 function square(): number;
13 function square(n: number): number;
14 function square(n?: number) { return n * n; }
15 const n = 2;
16 const result = n * n;
```

Listing 2.48: Inline function - overloading

2.8.10. Inline method

Just like functions, methods can also be inlined. When inlining a method, there are two cases to be considered: inlining within the same class and inlining across class borders. One of the main problems is the member-access with the `this` keyword. Listing 2.49 illustrates this problem. The `Car`'s method `drinkAndDrive` is inlined into the `Driver`'s method. It references its class' own method `drive` with `this.drive();`. If the code is copied as is, this would change the behavior, because the `Driver`'s `drive` method would be called, which may or may not be defined. Thus, `this` must be replaced with `this.car` or just `car` if it was not a property.

```
1  // before
2  class Car {
3    drive() { return "vroom"; }
4    drinkAndDrive() { return "glug " + this.drive(); }
5  }
6  class Driver {
7    car: Car = new Car();
8    drinkAndDrive() { return this.car.drinkAndDrive(); }
9  }
10
11 // after
12 class Car {
13   drive() { return "vroom"; }
14 }
15 class Driver {
16   car: Car = new Car();
17   drinkAndDrive() { return "glug " + this.car.drive(); }
18 }
```

Listing 2.49: Inline method

2.9. Refactoring: String concatenation to template literals

This refactoring converts a string concatenation to a string interpolation and the other way around. First, the template literal is analyzed to gain deeper insight. Afterwards, each case is examined individually.

2.9.1. Template literals

Template literals allow to write multi-line strings and to evaluate strings containing placeholders. Instead of single or double quotes, template literals are enclosed by backticks `` `...` ``. A placeholder consists of a dollar sign and an expression enclosed by curly braces `${expression}`.
It is also possible to place a tag function before the template string, which gives the opportunity to pre-process the template literal. As illustrated in listing 2.50, an array of all string literal sections are passed as the first argument and all the values of the placeholders are passed as the remaining arguments. [31]

```
const age = 12;
const interpolated = `Tom is ${age} years old`;
const newLine = `wait for the break
booom, here is the the new line`;

function tagArrow(literals: TemplateStringsArray,
          ...placeholders: number[]) {
    let output = "";

    for (let i = 0; i < placeholders.length; i++) {
        output += literals[i];
        output += "-> ";
        output += placeholders[i];
    }

    return output;
}

const taggedTemplate = tagArrow`Peter ${20}, Rachel ${27}`;
// Peter -> 20, Rachel -> 27
```

Listing 2.50: Template literal example

2.9.2. Cases

This refactoring must be invokable from any context, whether it is a variable declaration or a function call.

As function argument

In listing 2.51, the function expects a string and a string concatenation is provided. In such situations, this refactoring must offer the action *To template literal*.

```
//before
console.log("foobar is " + 32 + " years old");

//after
console.log(`foobar is ${ 32 } years old`);
```

Listing 2.51: As function argument

As variable declaration

This is the other base case, where a string concatenation is assigned to a variable. Even in such situations, this refactoring must offer the option *To template literal*.

```
1 // before
2 const age = 42;
3 const foo = "foobar is " + age + " old";
4
5 // after
6 const age = 42;
7 const foo = `foobar is ${age} old`;
```

Listing 2.52: As variable declaration

Remove parentheses

If an expression is parenthesized in a string concatenation, then the parentheses are stripped away for the template literal as demonstrated in listing 2.53.

```
1 // before
2 const foo = "foobar is " + (42 + 5) + " old";
3
4 // after
5 const foo = `foobar is ${ 42 + 5 } old`;
```

Listing 2.53: Remove parentheses

Arithmetic expression

The evaluation of an arithmetic expression depends on the expression's location in a string concatenation. Any arithmetic expression prior to the first string is evaluated normally and therefore, it can contain any arithmetic operators as shown in listing 2.54. Arithmetic expressions after the first string are evaluated based on their relative operator precedence as demonstrated on line 4 and onward.

```
1  // before
2  const foo1 = 4 + 4 + " Result"; // 8 Result
3  const foo2 = 5 - 4 + " Result"; // 1 Result
4  const foo3 = "Result " + 4 + 4; // Result 44
5  const foo4 = "Result " + 5 - 5; // Error
6  const foo5 = "Result " + 5 * 5; // Result 25
7
8  // after
9  const foo1 = `${4 + 4} Result`;
10 const foo2 = `${5 - 4} Result`;
11 const foo3 = `Result ${4}${4}`;
12 // refactoring for foo4 not available
13 const foo5 = `Result ${5 * 5}`;
```

Listing 2.54: Arithmetic expression

New line

When a string concatenation contains a line feed character `\n`, then it must be converted to an actual new line in template literal as illustrated in listing 2.55.

```
1 // before
2 const foo = "wait for new line\n"
3 + "bada bum!";
4
5 // after
6 const foo = `wait for new line
7 bada bum!`;
```

Listing 2.55: New line

Escape backtick and dollar

Backticks and dollar signs with opening brace must be escaped before the conversion as shown in listing 2.56.

```
1 // before
2 const foo = "I contain a` backtick and a dollar ${}";
3
4 // after
5 const foo = `I contain a\` backtick  and a dollar $\\{}`;
```

Listing 2.56: Escape backtick and dollar

Escape sequences

Klaus Meinhardt has pointed out in pull request that octal escape sequences are not allowed in template literals. [24] Therefore, when a string contains an octal escape sequence, the octal sequence must be converted to unicode characters as demonstrated in listing 2.57.

```
1 // before
2 const foo = "Unicode \u0023 \u{0023} " + "Hex \x23 " + "Octal \43";
3
4 // after
5 const foo = `Unicode # # Hex # Octal #`;
```

Listing 2.57: Escape sequences

To string concatenation

When converting to string concatenation, all expressions must be placed in the right position of concatenation. In such situations, this refactoring must offer the

option *To string concatenation*.

```
1 // before
2 const age = 42;
3 const foo = `foobar is ${age} old`;
4
5 // after
6 const age = 42;
7 const foo = "foobar is " + age + " old";
```

Listing 2.58: To string concatenation

Add parentheses

When a placeholder contains a compound expression, then this expression must be parenthesized before converting to string concatenation as shown in listing 2.59.

```
1 // before
2 const foo = `foobar is ${ 42 + 5 } old`;
3
4 // after
5 const foo = "foobar is " + (42 + 5) + " old";
```

Listing 2.59: Add parentheses

Nested template literals

As demonstrated in listing 2.60, if a template literal contains another template literal, just the selected template literal is transformed. In other words, embedded expressions' template literals will not be converted recursively. In case the outer template literal is selected, then only outer one is converted as show on line 6. In case the inner template literal is selected, then only inner one is converted as shown on line 10.

```
1  // before
2  const foo = `foobar is a
3      ${ `grown-up ${ age > 60 ? "and needs assistance" : ""}` }`;
4
5  // after outer
6  const foo = "foobar is a " +
7      `grown-up ${ age > 60 ? "and needs assistance" : ""}`;
8
9  // after inner
10 const foo = `foobar is a
11     ${ "grown-up " + (age > 60 ? "and needs assistance" : "")" }`;
```

Listing 2.60: Nested template literals

Tagged templates

As shown in listing 2.61, when a template literal is used in combination with a tag function, then this refactoring must not be provided because the tag function can only be used with a template literal.

```
1 // before
2 function myTag (literals: TemplateStringsArray, ...exprs: any[]) {}
3 const foo = myTag`foobar ${4 + 6} rocks`;
4
5 // refactoring not available
```

Listing 2.61: Tagged templates

3. Design

This chapter contains all the design decisions made for each refactoring or quick-fix.

3.1. Quick-Fix: Interface stubbing for object literal

This section describes the design decision made for this quick-fix. First, the general procedure is explained. Afterwards, each case will be discussed in more detail. Since the cases *as function argument*, *inheritance* and *partially existing and nullable members* are dealt with in general procedure, these cases will not be discussed separately.

3.1.1. General procedure

There are four major steps to this quick-fix.

1. **Get interface's declaration**

   Since this quick-fix is available for variable declarations and function calls, there are two starting points.

   If this quick-fix is triggered from a variable declaration, the interface's declaration is simply extracted from the variable's type.

   In case of a function call, it is more involved. Since the type information for arguments is not available in a function call, the function declaration must first be retrieved. Thereafter, the right parameter is fetched based on the concerning argument's position. Subsequently, the interface's declaration is extracted from the parameter's type.

2. **Fetch all members**

   All members are fetched from the concerning interface and inherited interfaces/classes. The next step is to filter out existing members and optional properties.

3. **Create missing members**

   For creating a property, two pieces of information are needed: a name for the property and an expression. The symbol, which was fetched in the previous step, contains the relevant information. One is the name, which is directly reused for the creating the property. The other one is its type, which is indirectly used for creating of expression. How the expressions are created will be explained in the following subsections because it is handled for each type differently.

4. **Manipulate AST**

   The new members will be inserted at the start of the corresponding object literal.

3.1.2. Basic types

The default values of basic types can be created using a lookup table such as table 3.1.2. Since the type `any` can have any value, it was decided that the default value is `"any"` so that it can be distinguished from others at a glance.

Type	Default value
string	`""`
number	`0`
boolean	`false`
null	`null`
any	`"any"`
array	`[]`

Table 3.1.: lookup table for basic types

3.1.3. Objects

There are two options to deal with classes. One is to always call their default constructor. The other is to call the existing constructor and to fill it with random data. The latter poses the danger of misuse. That is why it was decided to always call the default constructor.
In case there is no default constructor available, the checker notifies the user that the constructor is missing arguments. The default constructor is called for primitive wrapper objects as well.
In order to distinguish the type `object` (lowercased) from the type `Object` (uppercased), it was decided that the argument `"anyObject"` is passed to the constructor for the type `object` (lowercased).

If type is an object literal, step 4 *Create missing members* will be triggered using members of the object literal. Thanks to this recursion, nested object literals can be created.

3.1.4. Methods

An arrow function can be assigned to a method signature or property with a function type, whereas a method can only be assigned to a method signature. In order to keep the complexity level low, it was decided to create an arrow function in either case.
First, relevant attributes like parameters and return type are collected. Subsequently, a new arrow function is created with collected attributes as well as a new function body containing a throw expression.

3.1.5. Intersection and Union Types

In case of a union type, the first type is picked. Thereafter, a default value is provided for the picked type.
In case of an intersection type, all types are fetched from the intersection. Subsequently, all properties are retrieved from those types. After that, these properties are combined into one pool of properties, making sure there are no duplicates. Finally, a default value is provided for each property.

3.1.6. Tuples

The encapsulated types are extracted from tuple type. Afterwards, a default value is provided for each encapsulated type.

3.1.7. Type Alias

If a type alias is different from a real type such as intersection or union types, the real type is revealed with `TypeChecker`.

3.1.8. Generics

As long as functions and classes copy the type argument for generic correctly, there will be no trouble.

3.2. Refactoring: Lambda to function

This section discusses the design decisions made for the refactoring *Lambda to function*.
To implement this refactoring, the AST must be modified. These modifications vary based on the cases discussed in the analysis chapter.

3.2.1. To named function

The lambda expression node is nested in a variable node. The hierarchy must be traversed to get all the needed information. Furthermore, the correct node needs to be replaced. The replacement needs to happen on the statement-level. An illustration of the corresponding fields can be seen in figure 3.1. Name, parameters, and return type can be copied as is depending on implementation. The body needs to be converted if it is an expression. In that case, a statement block with a single return statement containing the expression should be created.

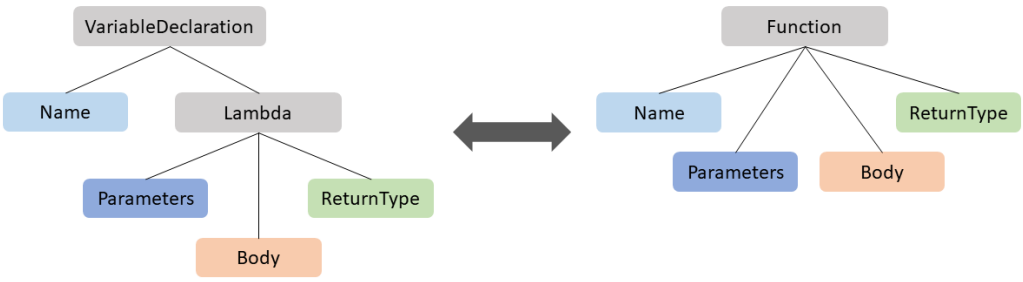

Figure 3.1.: Lambda declaration to named function

In case the lambda declaration is part of a multi-declaration line, the variable declaration containing the lambda expression must be removed from the list instead of deleting the whole statement node. In that case, a new function declaration node must be inserted after that statement node as seen in listing 3.1:

```
1 // before
2 const a = 3, b = n => n * 2;
3
4 // after
5 const a = 3;
6 function b(n) {
7   return n * 2;
8 }
```

Listing 3.1: Multi-declaration

3.2.2. To anonymous function

In this case, the substitution is simpler. The lambda expression is replaced by a function expression as seen in figure 3.2. The function's name is left empty. The body conversion is handled the same as in the previous case.

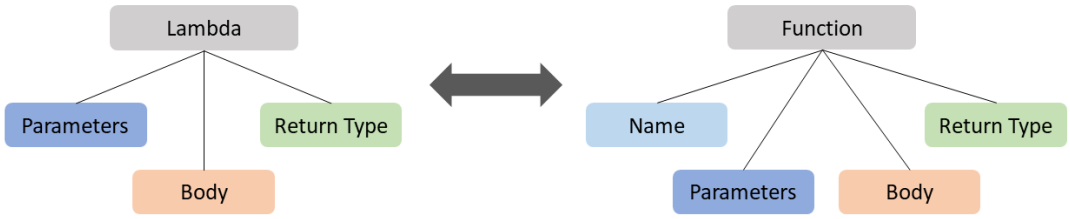

Figure 3.2.: Lambda expression to anonymous function

3.2.3. To arrow function

This is the reverse case of the previous one. The same considerations must be made here with exception of the body. A lambda's body can be either an expression or a statement block. A simple solution would be to just copy the block and be done with it. However, if that function has a singular return statement, it is converted to an expression body.

3.3. Refactoring: Inline variable

This section discusses the design decisions made for the refactoring *inline variable*. To implement this refactoring, the AST must be modified. These modifications vary based on the cases discussed in the analysis chapter.

3.3.1. General behavior

There are three steps to this refactoring.

1. Check if refactoring applicable
2. Find references with the help of symbols
3. AST manipulation

As laid out in the analysis chapter, there are two basic cases when the refactoring may be applied. When a local variable declaration or its usage is selected.
Three pieces of information are necessary to execute the refactoring:

- the variable declaration
- the references to – or usages of the variable
- (optional) the selected usage, if invoking `Inline here`

The second step of the refactoring requires finding all references in the outermost scope, where the variable is visible. This is done by finding which scope the declaration is in. Usages of the variable may be in a nested scope, so it is dangerous to use them to set the scope.
Once all references have been found, the third step is to manipulate the AST.

Inline all

In the case of `Inline all`, as the name suggests, all usages must be replaced. Then, the declaration can be removed.

Inline here

In the case of `Inline here`, only the selected usage is replaced. If there are no other usages left, the declaration is removed.

3.3.2. Restrictions on refactoring

As discussed in the analysis, there are three cases where the refactoring should not be invoked but should still be initially provided for transparency.
The current refactoring interface provides two methods, of which `getEditsForAction` executes the refactoring. It should report the failure accordingly.
In its current state, however, the interface does not support this behavior. There is no way to send a refactoring report message. The only way to prevent the refactoring is to not provide it at all.

3.4. Refactoring: Inline function

This section discusses the design decisions made for the refactoring *Inline function*. To implement this refactoring, the AST must be modified. These modifications vary based on the cases discussed in the analysis chapter.

3.4.1. General behavior

There are three steps to this refactoring.

1. Verify that refactoring is applicable
2. Find references in file with the help of symbols
3. Manipulate AST

As laid out in the analysis chapter, there are two basic cases when the refactoring may be applied. When a function or method declaration is selected, or its call. Three pieces of information are necessary to execute the refactoring:

- The function or method declaration
- The function or method calls respectively
- The selected call, if invoking *Inline here*

When transforming the AST, the function-call node is the anchor. The function's statements are inserted before that node's parent statement. The same goes for any parameters, which need to be converted to local variable declarations.
If the call node's parent is a statement, that statement is deleted. Otherwise, the call node is replaced with the function's return expression.

3.4.2. Multiple return statements

If the function has multiple return statements and its type is not `void`, a return-value variable declaration is inserted before the function's statements. Furthermore, the call node is replaced with that variable.

3.4.3. Name conflict

In order to resolve name conflicts, certain steps need to be taken:

1. Determine all names in the target location

2. Compare them with names in the function

3. Replace conflicting names with unique ones

Special care needs to be taken for object literal's properties. If the shorthand is used, it must be replaced with the longer notation, `{ property_name: variable_name }`, as seen in listing 3.2. Reason being that `arg` is the property's name. Simply replacing it with `arg_1` would make the compiler retrieve a different property, which would change behavior.

```
1  // before
2  function foo({arg}) {
3      return arg;
4  }
5  function bar(arg: number) {
6      return arg * foo({ arg: 21 });
7  }
8
9  // after
10 function bar(arg: number) {
11     const { arg: arg_1 } =  { arg: 21 };
12     return arg * arg_1;
13 }
```

Listing 3.2: Destructuring name conflict

3.4.4. Restrictions on refactoring

As discussed in the analysis chapter, there are cases where the refactoring should not be invoked but should still be initially provided for transparency.
The current refactoring interface provides two methods, of which `getEditsForAction` executes the refactoring. It should report the failure accordingly.
In its current state, however, the interface does not support this behavior. There is no way to send a refactoring report message. Thus, the only way to prevent the refactoring is to not provide it at all, which is done in the `getAvailableActions`-function.

3.5. Refactoring: String concatenation to template literals

This section describes the design decisions made for this refactoring. First, the template expression is explained. Thereafter, the general procedure is described. Afterwards, each case will be discussed in more detail. Since the cases *as function*

argument, *as variable declaration* and *to string concatenation* are dealt with in general procedure, these cases will not be discussed separately.

3.5.1. Template expression structure

As illustrated in figure 3.3, a template expression is made of two parts: a template head and template spans. A template span either have a template middle or a template tail. A template span must have a string and an expression, whereas a template head has only a string.

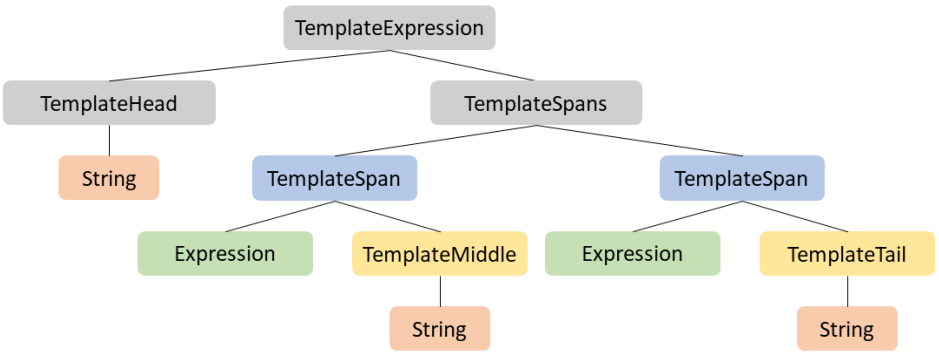

Figure 3.3.: Template expression structure

3.5.2. General procedure

Since this refactoring offers two actions: *To template literal* and *To string concatenation*, there are two general procedures.

To template literal

1. **Get top binary expression**

   A string concatenation consists of nested binary expressions. The binary tree is traversed up until top binary expression is reached. Thereafter, the top binary expression is returned.

2. **Transform tree to array**

   Element access is cumbersome in a tree. Therefore, the tree is first converted to an array of nodes with correct order.

3. **Create template literal**

   Template head and tail need to be created individually. The template middle parts are created for each of the remaining nodes in the array.

4. **Replace node**

   Finally, the top binary expression node is replaced by the template literal.

To string concatenation

1. **Create array of expressions**

   An array of expressions is created based on the template literal.

2. **Transform array to tree**

   An binary tree with correct traversal order is created based on the array of expressions.

3. **Replace node**

   Finally, the template literal is replaced by the newly created binary expression tree.

3.5.3. Remove parentheses

If an expression is wrapped in parentheses, then the expression is extracted.

3.5.4. Arithmetic expression

All arithmetic expressions prior the first string are considered and handled as single expression. This ensures that the whole expression will be calculated correctly.

3.5.5. New line

A preliminary study has been done on whether it is possible to explicitly set a new line in a template literal. But it appears that the string writer does not support the explicit setting of a new line. That is why this case will not be implemented.

3.5.6. Escape backtick and dollar

The backtick is escaped by inserting a backslash in front of it. The dollar sign can be escaped in two ways: putting a backslash in front of the dollar sign or putting a backslash between the dollar sign and the opening brace. There is no behavioral difference, thus it was decided to use the latter option.

3.5.7. Escape sequences

If a string contains a backslash and a sequence of numbers from 0 to 7, then this octal escape sequence must be converted to a unicode character. First, the octal number must be converted to hexadecimal. Finally, a unicode character is created based on the hexadecimal value.

3.5.8. Add parentheses

If the placeholder of a template literal has a compound expression, then parentheses are added to the compound expression.

3.5.9. Nested

From the selected token, the first template literal ancestor in the node hierarchy is chosen for the conversion to string concatenation.

3.5.10. Tagged templates

If the parent of a template literal is a tag function, then this refactoring is not provided.

4. Implementation

This chapter shows how all refactorings and quick-fixes are effectively implemented.

4.1. Quick-Fix: Interface stubbing for object literal

4.1.1. General

Retrieving the interface's declaration from a variable is very simple. In case of a function call, it requires some additional steps, but it is still easy. And thanks to TypeChecker all properties can easily be retrieved from the relevant interface and its inherited classes/interfaces with just one method call.
As the design part has shown, when a method is required, an arrow function is provided instead of a method implementation. Thanks to this insight, all member kinds for object literal can be considered as properties without any change in external behavior. Thus, further branch divergence could be avoided.
Before creating a property or expression, the syntax kind may need to be redirected in order to provide the right default value. Difficulties lied in redirecting the syntax kinds without disturbing already working implementations.

4.1.2. Corner cases

Redirect interface

If a property's type is an interface, it is desired that an object literal is produced. But if a property's type is an primitive wrapper objects, it is expected that a `new`-expression is created. Both have the type interface because the wrapper's functionalities are defined in interface. Since the wrapper's declarations are not all interfaces, this knowledge is leveraged to separate them from each other.

Redirect union

Booleans and enums are represented as union types internally. Therefore, the first type from union is only picked as long as the the property is not boolean and not enum.

Redirect anonymous object

When a function or object literal is behind a type alias, then they have the type anonymous object. In such cases it is first checked whether it can be a function or not. Afterwards, it is redirected to the right syntax kind accordingly.

Generic class

Normally, a class has the type class. But the generic class has the type reference. Therefore, when creating the default expression for a class instance, it also is checked, whether it can be a reference type.
Before creating a `new`-expression for a class, an identifier is needed. The identifier is created based on a string. The function `typeToString()` of `TypeChecker` is used to create the string. In case of a generic class, the returned string still contained the angle brackets with type argument. Thus, the identifier could not be created. There is a filter before creating the identifier that ensures the angle brackets with its content are removed.

Intersection

Since an intersection type can be behind type alias, it is checked, whether it can be an intersection type or not. Thereafter, it is accordingly redirected to syntax kind intersection.
Retrieving the types from an intersection works fine for most cases. However, there is one particular case, which needs a special treatment. This situation arises only, if an intersection type is behind a type alias and this type alias is inside a tuple. For that, a flag named `wasRedirectedForIntersection` was created to retrieve the types differently.

4.2. Refactoring: lambda to function

As the analysis has shown, only three of four considered cases are necessary. Therefore, only these three cases are discussed in this section.

4.2.1. To anonymous function

This action is visible for every lambda expression independent of surrounding code. That means it does not matter, if the lambda expression is meant to be a function argument or variable declaration. In order to invoke a refactoring, source code must be selected. Whether this refactoring is provided or not depends only on the starting point of the selection. Thus, to invoke this code action, the selection must

begin with the head of the lambda expression. But if the selection begins inside the body of the lambda expression, then this code action will not appear because it already is a different code context.

Procedure

Almost every attribute is copied from the ArrowFunction node into a new *FunctionExpression* node without any modification. As demonstrated in figure 4.1, if the body of the ArrowFunction is a single expression, then a new ReturnStatement will be created where the extracted expression is inserted. After that a new FunctionBlock will be created with the ReturnStatement. If the body of the ArrowFunction is already a FunctionBlock, then it will be copied without modification. The name of the new FunctionExpression node is left empty. Thereafter, the old ArrowFunction node is replaced by the new FunctionExpression node.

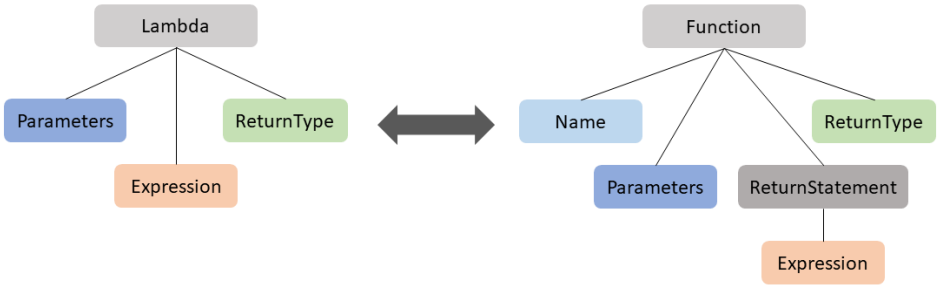

Figure 4.1.: AST transformation with single variable

Corner cases

The comments are not a part of the AST. In order to handle comment manipulation, there are helper functions available. When a single expression is transformed to a function block, as shown in listing 4.1, the multiline comment cannot be preserved despite the helper functions. The reason for this behavior is unknown as it has not been pursued. Since this is an unusual location for writing a comment, no further steps are taken to fix this particular issue. In other situations the comments are preserved.

```
1 # Before
2 const foo = () => /* comment */ "bar";
3
4 # After
5 const foo = function(){
6   return "bar";
7 }
```

Listing 4.1: difficult comment

4.2.2. To named function

This action is only available if the lambda expression is defined as a variable declaration. It can be invoked, if lambda expression or variable declaration is partially selected. In case of single variable declaration, it does not matter if the variable name or the keyword like `const` is selected. But in case of multiple variable declaration, it works only if the corresponding variable name of the lambda declaration is selected.

Procedure

Most attributes from ArrowFunction are used to create the new FunctionDeclaration node. As shown in figure 4.2, the modifiers like `export` are taken from VariableStatement node. The name is extracted from the corresponding VariableDeclaration node. Like *To anonymous function*, if the body of ArrowFunction is a single expression, then the Expression is transformed to a Block with a single ReturnStatement. In case of single VariableDeclaration, the VariableStatement node is replaced by the newly created FunctionDeclaration node.
In case of multiple VariableDeclaration, the relevant VariableDeclaration node is removed and the newly created FunctionDeclaration node is inserted right after VariableStatement as demonstrated in figure 4.3.

4.2.3. To arrow function

This action is visible for every function expression as long as the name is not used. To invoke this action, function expression must be partially selected and the selection must begin with header of function. If the selection starts inside the body, then this action will not appear because it already is a different code context.

Procedure

Apart from body, every attribute is copied from FunctionExpression node into new ArrowFunction node. If the body of FunctionExpression contains only one

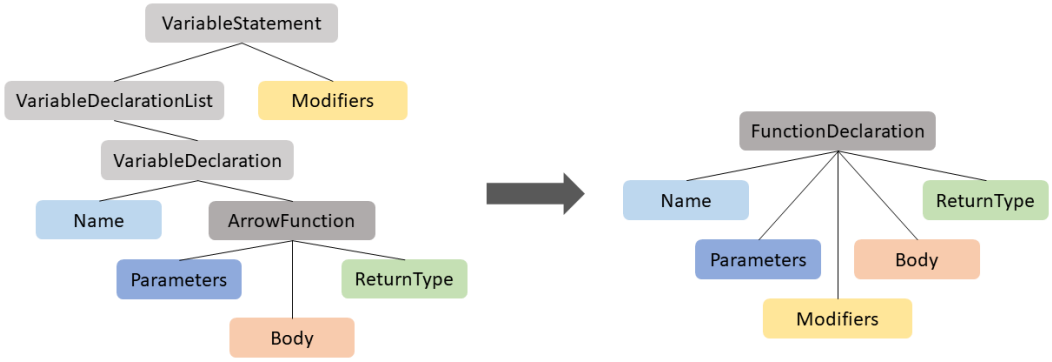

Figure 4.2.: AST transformation with single variable

statement and this statement is a ReturnStatement, then the expression of the single statement will be used as the new body for ArrowFunction node. In other situations the body is just copied without modification to new ArrowFunction node. After that the old FunctionExpression node is replaced with the new ArrowFunction node.

Corner Cases

When the body of FunctionExpression is transformed into a expression, the comments from FunctionExpression will be lost because comment is not a part of the AST. Fortunately, there is a helper function which can copy comments from one node to other node.
In case the body of FunctionExpression is a single ReturnStatement, it is necessary to be verified that the expression of the ReturnStatement is not empty. Only then it is permissible to transform into a expression.
As demonstrated in listing 4.2, in JavaScript the name of function expression can only be used inside the body like for recursion purpose. If the name is used, then this refactoring will be not provided.

```
1 [1,2,3].map(function factorial(n): int {
2     return n <= 0 ? 1 : n * factorial(n - 1);
3 });
4
5 factorial(5); // [ts] Cannot find name 'factorial'
```

Listing 4.2: function expression name usages

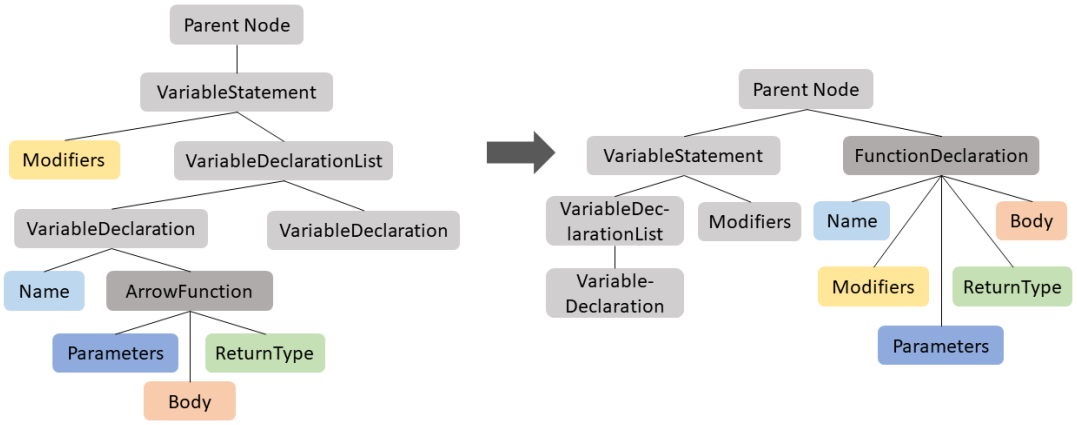

Figure 4.3.: AST transformation with multiple variables

4.3. Refactoring: Inline variable

There are two basic cases when the refactoring may be applied. When a local variable declaration or its usage is selected. The former is true if the parent node of the selected token is a `VariableDeclaration` and it is in a `VariableStatement`.
The identifier's symbol has a property named `valueDeclaration`. This is the symbol's declaration node. If this passes the same test as above, we know it is a local variable.
To find all usages of the variable, `FindAllReferences.Core.EachReferenceInFile` was used as a first measure. This did not work, however, because it never ran through more than one reference. Therefore, a custom solution was implemented: `getReferencesInScope`.
Regarding the operator precedence, the compiler API does not provide a helper function to assess if parentheses are necessary in a general case. There is however a `parenthesizeBinaryOperand` function, which can decide whether to parenthesize the expression, based on the target node being part of a binary expression or not. Other cases need to be handled manually according to the points made in the analysis chapter.
For `Inline all`, an extra step needs to be taken, when replacing the node. Since each node must have a unique id, just copying the expression node and passing it along is not enough. The id needs to be replaced. Unfortunately, we did not find any general enough copy-function that could copy any node and assign a new unique id. To remedy this, a function called `makeIdUnique` was made. It uses the API function `getNodeId`, which assigns a unique id if id is undefined. `makeIdUnique`

uses this knowledge and assigns undefined to the node's id-property and calls `getNodeId`. This is a hack and if a better solution is found in the future, this should be rewritten.

4.4. Refactoring: Inline function

This section talks about implementation details for the refactoring *Inline function* that may not be immediately evident and are not explained in the code.

canInline Inlining only makes sense if a function has a body and if that body is not empty. Thus, appropriate checks are performed.

getCallsInScope When inlining a method a simple symbol comparison is not enough. Because TypeScript is a prototype-based language, object members can be modified after instantiation. Thus, a method `drive` could be rewritten to do something else than a class `Car`, of which the object is a member, defines. This means that a class instance's member cannot have the same symbol as the class' member. That is why the indirection via an object's type is necessary.

getInlineInfo To transform a function body, `getInlineInfo` uses the visitor pattern. The visitor is the function called `transformVisitor`. It needs to be an inner function because it only accepts one argument, but requires contextual information, which is fed in using closure.

4.4.1. Inlining across multiple files

According to Microsoft's specifications, the function `getAvailableActions` must be very quick. To our knowledge however, the TypeScript compiler API does not provide a quick and efficient way to search a file for references to a symbol. Therefore, the only way is to traverse all nodes in every file to find references. This may take a very long time for large code bases. But due to the current lack of feedback in `getEditsForAction`, it must be determined whether the refactoring should be provided in `getAvailableActions`. Thus, it was decided to only provide inlining within a file. This should not be too much of a limitiation. Judging from personal experience, this refactoring is rarely used across multiple files.

4.4.2. Overloading

Currently, overloaded functions are not supported. This is not too complex to implement, though. `Symbol` has a member `declarations`, which contains all dec-

larations. If iterating over that array results in a function declaration which is applicable for the refactoring, it is chosen.

4.5. Refactoring: String concatenation to template literals

This section talks about implementation details for the refactoring *String concatenation to template literals* that may not be immediately evident and are not explained in the code.

4.5.1. Transforming tree to array

The array of expressions is built while the binary tree is traversed in-order. As long as no string literal is visited for the first time, the visited expressions are considered as a single expression. During the traversal, if an expression is a parenthesized expression, the inner expression is extracted.

4.5.2. Transforming array to tree

A binary expression tree is needed for creating string concatenation. A binary tree is built using an array of expressions. The binary tree is built in a left-associative manner with plus operators.

4.5.3. Creating template literal

As shown in design, the template expression structure expects that a string is followed by an expression and an expression is followed by a string. In other words, a string cannot be followed by another string.
That is problematic when a string concatenation is made of consecutive strings. The solution is to consume consecutive strings in a greedy manner. In other words, the string is concatenated with the next string internally while as the next expression is a string literal. Subsequently, the string is inserted in a template part.

4.5.4. Decoding raw string

For handling octal escape sequence, it is required to retrieve the raw string because the `text` property from string literal nodes is already interpreted. The raw string is retrieved with the node's method `getText()`.
First, the single or double quotes must removed from raw string. Apart from the octal escape sequence, the unicode and hexadecimal escape sequence must also

interpreted to a unicode character, since the interpretation of the raw string is done manually.

5. Conclusion

The following code actions have been implemented:

- *Inline local* – Pull-request: #28522
- *Inline function* – Pull-request: #29096
- *Convert lambda to function* – Pull-request: #28250
- *Convert string concatenation to template literal* – Pull-request: #28923
- *Interface stubbing for object literal* – Pull-request: #28863

At the time of writing this document, the pull-request for *Convert lambda to function* has been reviewed and approved by Microsoft. Others have not been reviewed yet.
Initially, we anticipated that we could implement more code actions. However, unfamiliarity and other unforeseen problems made it more difficult. What is more, the split focus between refactorings and quick fixes drained additional resources. This was due to there being significant differences in their APIs, which were not documented well. This forced us to invest a lot of time to understand how to use them. In hindsight, we should have focused only on refactorings to be able to contribute more. Lack of documentation was a general problem, so there was always a need to invest a lot of time into understanding how to use the various APIs, i.e. the compiler's AST API. One more reason why it took longer for each feature was our initial limited understanding of TypeScript. Neither of us had a lot of experience with TypeScript. We learned very much about the language itself while working on this project.
On a more positive note, the internal code reviews improved the final code quality significantly. Some even revealed bugs. Doing code reviews was certainly worth the time investment. Writing test before implementing a feature, as suggested by Thomas Corbat, also proved very helpful while refactoring.

5.1. Outlook

Since not all code actions have been reviewed yet, it is expected that there will be additional work cleaning the code. Since the repository is managed by Microsoft,

it is not possible to say when this work will be finished. Thus, it does not lie in the scope of this project.
The implemented quick-fix, *Interface stubbing*, makes use of existing error messages. However, just before opening the pull-request, one of the error messages did not appear anymore. A related code change in the master branch seems to be causing this. Therefore, issue #28767 has been opened on Github regarding this matter. [32]
Apart from that, there is also the matter of the design flaw in the refactoring API, which prevents the language server to send diagnostic information to the client in case a refactoring fails. This would be useful for transparency. For example, the user should know why inlining a function fails, when the function has an empty body. The user expects the refactoring to be possible, but currently, the refactoring is just not available. Instead, the refactoring should be shown initially, but if the command is invoked, an error message should tell the user that the function cannot be inlined because the function body is empty. An appropriate issue has been openend on Github. [16]
Lastly, overloaded functions have been overlooked for the refactoring *Inline function* as mentioned in section 4.4.2.

Glossary

API Application Programming Interface. viii, ix, 4, 5, 9, 15, 19–21, 48, 71, 72, 75, 76

AST The Abstract Syntax Tree is a tree that represents the abstract syntactic structure of source code. 14–16, 19, 20, 22, 56, 58–61, 68, 70, 71, 75

JSON JavaScript Object Notation. 10, 12, 14, 24, 77

JSON-RPC is a remote procedure call protocol encoded in JSON. It is a very simple protocol, defining only a few data types and commands [39]. 9, 10, 77

LSP The Language Server Protocol is an open, JSON-RPC-based protocol for use between source code editors or integrated development environments and servers that provide programming language-specific features[40]. ix, 8–12, 14

mixin The mixin concept allows a class to add more functionality without making use of inheritance's specialization . In other words, a class implements an interface with implemented methods [41]. 25

Bibliography

[1] *Advanced Types*. Microsoft. URL: https://www.typescriptlang.org/docs/handbook/advanced-types.html (visited on 12/02/2018).

[2] *Allgemeine Infos Diplom*. HSR. URL: https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html (visited on 09/28/2018).

[3] *Archive*. HSR. URL: https://archiv-i.hsr.ch/ (visited on 09/28/2018).

[4] basarat. *TypeScript Compiler Internals*. URL: https://basarat.gitbooks.io/typescript/docs/compiler/overview.html (visited on 10/19/2018).

[5] *Build cross platform desktop apps with JavaScript, HTML, and CSS*. Electron. URL: https://electronjs.org/ (visited on 12/18/2018).

[6] *Coding guidelines*. Microsoft. May 11, 2018. URL: https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines (visited on 10/23/2018).

[7] *Convert lambda expression declaration to function issue #23299*. URL: https://github.com/Microsoft/TypeScript/issues/23299#issuecomment-429288585 (visited on 10/24/2018).

[8] *Debugging Language Service in VS Code*. Microsoft. URL: https://github.com/Microsoft/TypeScript/wiki/Debugging-Language-Service-in-VS-Code (visited on 12/10/2018).

[9] *Developer Survey Results 2018*. Stackoverflow. URL: https://insights.stackoverflow.com/survey/2018/ (visited on 12/18/2018).

[10] *Extending Visual Studio Code*. Microsoft. URL: https://code.visualstudio.com/docs/extensions/overview (visited on 10/14/2018).

[11] *Extensibility Principles and Patterns*. Microsoft. URL: https://code.visualstudio.com/docs/editor/refactoring (visited on 12/16/2018).

[12] *Extensibility Principles and Patterns*. Microsoft. URL: https://code.visualstudio.com/docs/extensionAPI/patterns-and-principles (visited on 10/14/2018).

[13] Martin Fowler. *Refactoring*. URL: https://refactoring.com/ (visited on 12/16/2018).

[14] Andy Hanson. *add support to convert lambda to function and vice-versa*. Nov. 7, 2018. URL: https://github.com/Microsoft/TypeScript/pull/28250%5C#issuecomment-436695129 (visited on 11/30/2018).

[15] Mohamed Hegazy. *TSServer: Use JSON-RPC as RPC protocol (discussion)*. Microsoft. Oct. 7, 2016. URL: https://github.com/Microsoft/TypeScript/issues/11423%5C#issuecomment-252105101 (visited on 10/18/2018).

[16] Giovanni Heilmann. *Extend tsserver interface to allow refactoring failure reporting*. Nov. 8, 2018. URL: https://github.com/Microsoft/TypeScript/issues/28410 (visited on 11/30/2018).

[17] Anders Hejlsberg. *Non-nullable types*. Microsoft. Feb. 18, 2016. URL: https://github.com/Microsoft/TypeScript/pull/7140 (visited on 10/23/2018).

[18] *How to Contribute*. Microsoft. URL: https://github.com/Microsoft/vscode/wiki/How-to-Contribute (visited on 12/10/2018).

[19] *Interfaces*. Microsoft. URL: https://www.typescriptlang.org/docs/handbook/interfaces.html (visited on 12/02/2018).

[20] Mickael Istria. *tsserver should implement the Language Server Protocol*. Sept. 30, 2016. URL: https://github.com/Microsoft/TypeScript/issues/11274 (visited on 10/15/2018).

[21] *Language Extension Guidelines*. Microsoft. URL: https://code.visualstudio.com/docs/extensionAPI/language-support (visited on 10/14/2018).

[22] *Language Server Protocol Overview*. URL: https://microsoft.github.io/language-server-protocol/overview.

[23] *Language Server Protocol Specification*. URL: https://microsoft.github.io/language-server-protocol/specification.

[24] Klaus Meinhardt. *add refactoring: string concatenation to template literals #28923*. Dec. 9, 2018. URL: https://github.com/Microsoft/TypeScript/pull/28923#discussion_r240027801 (visited on 12/09/2018).

[25] Microsoft. *Compiler Internals*. URL: https://github.com/Microsoft/TypeScript/wiki/Compiler-Internals (visited on 10/19/2018).

[26] Microsoft. *Creating Language Servers for Visual Studio Code*. URL: https://code.visualstudio.com/docs/extensions/example-language-server (visited on 12/20/2018).

[27] *Modulbeschreibung*. HSR. URL: http://studien.hsr.ch/allModules/19419_M_BAI.html (visited on 09/28/2018).

[28] *Object initializer*. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer (visited on 12/02/2018).

[29] Marius Schulz. *TypeScript 2.2: The object Type*. Feb. 24, 2017. URL: https://blog.mariusschulz.com/2017/02/24/typescript-2-2-the-object-type (visited on 12/14/2018).

[30] *Standalone Server (tsserver)*. Oct. 19, 2018. URL: https://github.com/Microsoft/TypeScript/wiki/Standalone-Server-(tsserver).

[31] *Template literals (Template strings)*. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals (visited on 12/02/2018).

[32] Arooran Thanabalasingam. *Missing Type_0_is_not_assignable_to_type_1 on variable declaration*. Nov. 30, 2018. URL: https://github.com/Microsoft/TypeScript/issues/28767 (visited on 11/30/2018).

[33] *TypeScript*. Microsoft. URL: https://github.com/Microsoft/TypeScript/ (visited on 09/28/2018).

[34] *TypeScript*. Microsoft. URL: https://github.com/Microsoft/TypeScript/issues (visited on 09/28/2018).

[35] *TypeScript - JavaScript that scales.* Microsoft. URL: https://www.typescriptlang.org (visited on 12/02/2018).

[36] *Visual Studio Code*. Microsoft. URL: https://github.com/Microsoft/vscode (visited on 09/28/2018).

[37] *Visual Studio Code - Code Editing. Redefined*. Microsoft. URL: https://code.visualstudio.com/ (visited on 09/28/2018).

[38] *What are the most common refactoring operations performed by GitHub developers?* Applied Software Engineering Research Group - UFMG. Jan. 4, 2017. URL: https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896b0db96d9d (visited on 12/14/2018).

[39] Wikipedia. *JSON-RPC*. URL: https://en.wikipedia.org/wiki/JSON-RPC (visited on 12/20/2018).

[40] Wikipedia. *Language Server Protocol*. URL: https://en.wikipedia.org/wiki/Language_Server_Protocol (visited on 12/20/2018).

[41] Wikipedia. *Mixin*. URL: https://en.wikipedia.org/wiki/Mixin (visited on 12/20/2018).

Appendix

A. Time evaluation

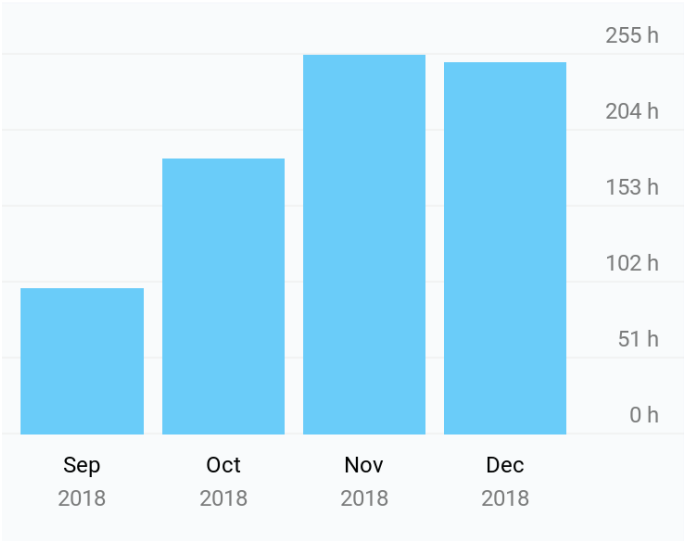

Figure A.1.: Time consumption per month

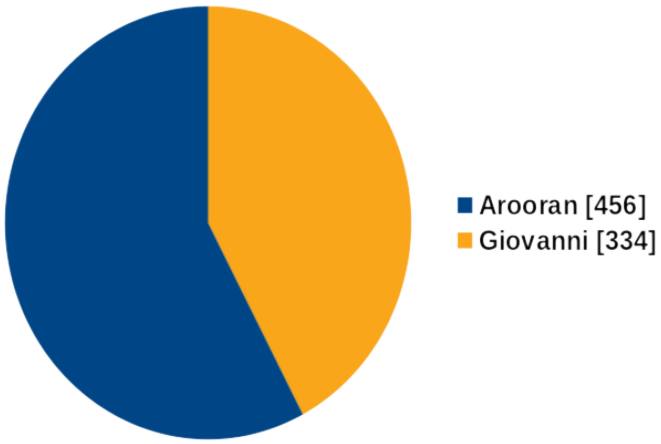

Figure A.2.: Time investment per person

Since testing was done as a part of implementation, the time consumption for testing is tracked in the implementation's time consumption. A major part of analysis and design was writing it down. That time was tracked as documentation. Only the actual process of analyzing and designing was marked as such.

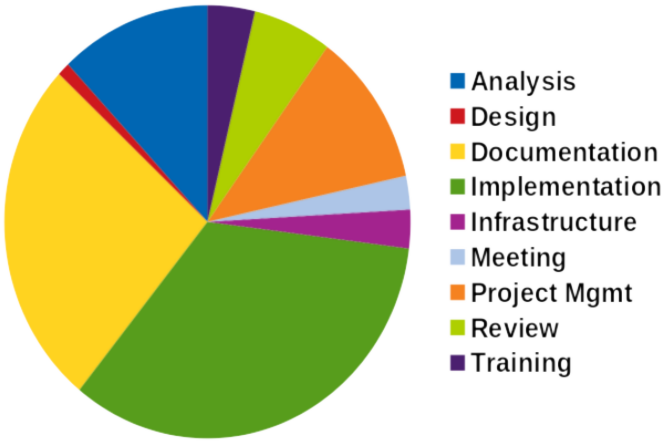

Figure A.3.: Time consumption per category

B. Project Plan

TypeScript Refactorings

Project Plan

Giovanni Heilmann
Arooran Thanabalasingam

Table of Contents

1. Introduction

1.1 Purpose

This document is intended to serve as a guideline for the project TypeScript Refactorings. Among other things, it contains time planning, a rough division of labor and quality measures. The project plan can be adapted during the project.

1.2 Scope

The project plan is valid throughout the project. Only the latest version is valid.

2. Project Overview

TypeScript is a programming language which is developed by Microsoft. VS Code is a source code editor which is open source and extensible. It runs on Windows, macOS and Linux. Among other things, it has support for debugging, syntax highlighting and code actions like refactorings and quick fixes. It is mainly written in TypeScript. Apart from TypeScript it supports many other languages.

As part of this project the TypeScript code base will be extended with further code actions.

2.1 Purpose and Goal

Priority	Issue No.	Description	Type
1	#16755	Interface stubbing	Quick-Fix
1	#18459	Inline local variable	Refactoring
1	#23299	Lambda to function	Refactoring
1	#27070	Inline method	Refactoring
2	#662	Reorder parameters	Refactoring
2	#18267	String concatenation -> template literals	Refactoring
2	#24827	Destructure function parameters	Refactoring
2		Move method to module/namespace	Refactoring
3	#16010	File capitalization	Quick-Fix
3	#22392	Create object from selected variables	Refactoring
3	#23552	Named parameter	Refactoring
3	#23830	Auto-import module as namespace	Quick-Fix
3	#23869	Type alias	Refactoring
3	#25175	Logic predicate	Refactoring
3	#25946	Var to destructuring	Refactoring
3	#26479	Extract function to outer scope	Refactoring

2.2 Scope of delivery

- software
- documentation
- project plan
- meeting minutes
- time evaluation
- personal reports
- declaration on the independent performance of the work

2.3 Assumptions and limitations

Each member has a fixed workload of 360 hours. Therefore, depending team's progress, the amount of implemented code actions may vary. These will be chosen based on the priority list. The list should provide enough work even if the team advances quicker than expected.

3. Project Organisation

The team consist of four people.

3.1 Organisation Structure

4. Management Processes

4.1 Weekly Meetings

In weekly meetings, the results of the last week will be presented and the work of the following week will be prioritized. These meetings take place each Thursday at 15:00.

4.2 Time Planning

Since this project is a bachelor thesis, the two programmers each have 360 hours availabe. That means a total of 720 working hours are available. The project runs from 17.09.2018 to 21.12.2018, which is a total of 14 weeks.

4.2.1 Estimated Plan

CW	PW	Milestone	Date	Description
38	1	project start	17.09.2018	
39	2			
40	3	LSP documentation		
41	4			
42	5			
43	6			
44	7	Prio 1 Code Actions		
45	8			
46	9	Prio 2 Code Actions		
47	10			
48	11			
49	12	code freeze		
50	13	abstract & poster	13.12.2018	finish abstract and release complete poster and send it to supervisor
51	14	submission	21.12.2018	submit paper

CW – calendar week
PW – project week

4.2.2 Actual Procedure

CW	PW	Milestone	Date	Description
38	1	project start	17.09.2018	
39	2			
40	3			
41	4			
42	5	Lambda to function		
43	6			
44	7			
45	8	Inline local		
46	9			
47	10			
48	11	Interface Stubbing		
49	12	Inline funciton Template literal code freeze		
50	13	abstract & poster	15.12.2018	finish abstract and release complete poster and send it to supervisor
51	14	submission	21.12.2018	submit paper

4.2.3 Task List

Description	estimated
document interface registerRefactor/registerCodeFix	2h
analyse syntax tree/symbol table	8h
document test infrastructure	2h
first refactor (together)	
analysis & design	16h
implementation & testing	16h
document refactoring	4h
review code	1h
review doc	1h
open pull-request (signing CLA)	1h
per refactor	
analysis incl. documentation	8h
design incl. documentation	8h
implementation & testing	8h
review code & discussion	2h
review doc & discussion	2h
sequence diagram with concrete example	8h
finalize documentation (formatting, content)	8h
design poster	4h
write abstract	4h
presentation preparation	24h

5. Infrastructure

VS Code	The source code editor supports TypeScript, which will be used for developing.
Dropbox	All documents are stored on Dropbox. Dropbox offers version control and change history.
Office Suite	Smaller documents are created with Office.
TeXstudio	The final documentation will be created with LaTeX. TeXstudio was chosen as the LaTeX editor because it indents whole paragraph.
Git & GitHub	Git is used as a version control system for source code. The repository is hosted on GitHub. Issues are also managed on GitHub.
Travis CI	Travis CI offers a free CI service for public repositories. On each change in the code base, an automated build process will be triggered with the corresponding execution of test cases.
Toggl	For time tracking and evaluation, the service of Toggle is used.
Waffle.io	Waffle.io displays Github issues on a clear board, which facilitates the project managment.

6. Quality Measures

This chapter describes measures and tools used to ensure the quality of the project.

6.1 Documentation

Documents are saved in a shared Dropbox folder. Dropbox offers a free service, where changed or deleted files can be restored for up to 30 days.

Documents written by one team member will be reviewed by the other.

6.2 Development

The source code is located on a public repository on GitHub.

Repository	https://github.com/D0nGiovanni/TypeScript

6.2.1 Procedure

Our development process looks roughly structured as follows:

1. Create a Feature branch per Use Case
2. Implement Use Case, including unit and integration tests
3. Create a Pull-Request
4. Review the code (incl. tests) by assigned team member
5. Discussion of possible discrepancies
6. If necessary: correct code back to step 4
7. Merge the Feature branch into Master branch

6.2.2 Code Reviews

Each issue will be assigend to a team member as developer. The other member takes on the role of reviewer and checks the code before merging into Master branch.

Found discrepancies are discussed between team members and any improvements resulting therefrom are defined and implemented.

6.2.3 Coding Guideline

Since code actions are contributed to TypeScript, following coding guideline must be considered:

https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines

6.2.4 Static Code Analysis

In order to detect errors early, a static analysis tool is used. Because Microsoft uses tslint for linting, tslint is chosen as static analysis tool.

6.3 Testing

6.3.1 Unit Test

Unit tests must be created for all essential functions. On one hand the tests are run manually by the developer before a commit and on the other hand tests will automatically be executed with each build. As a result, errors are detected early and can be corrected.

6.4 Definition of Done

- The code is
  - implemented
  - tested
  - committed in vcs
- The code review was conducted and accepted
- The code is documented (analysis, design, implementation)

C. Guides

1. Setup development environment

It is recommended to use a separate, isolated version of VS Code to run the tsserver. VS Code only has to be built once. Only the language server must be built upon each change. The following instructions shows how to run a separate VS Code instance and how to override the path to tsserver in VS Code. Parts of this section are taken from the TypeScript contribution guide [8] and from VS Code contribution guide [18].

1.1. Running development VS Code

First, ensure that all requirements of the following link are installed:
`https://github.com/Microsoft/vscode/wiki/How-to-Contribute#prerequisites`
Then, in the command-line, enter the following commands.

1. `git clone https://github.com/Microsoft/vscode.git`
2. `cd vscode`
3. `yarn`
4. `./scripts/code.sh` or `.\scripts\code.bat`
    - The first launch will take a while to build the entire VS Code.
    - The following times, it will reuse the built version and start more quickly.
5. Optional step: add the path `/vscode/scripts/` to environmnent variables or make a symlink to `scripts/code.sh`

1.2. Override tsserver path in VS Code

- Press CTRL+SHIFT+P
- Open the User Settinsg by typing "Open Settings (JSON)"
- Set the following entries in the User Settings of development VS Code:

```
{
    "typescript.tsdk": "</PATH/TO/REPO>/built/local",
    "typescript.tsserver.log": "verbose",
}
```

1.3. Open tsserver log

This works only if a TS file is active:

- Press CTRL+SHIFT+P
- Type "Open TS Server log"

1.4. Installing additional softwares

These additional applications are needed for testing.

- `npm install --global gulp`
- `npm install --global jake`

2. Building tsserver

- To build: `npm run build`
- When switching between Git branches, it is important to clean first and then build:
  1. `npm run clean`
  2. `npm run build`
- For hot-reloading:
  1. `npm run build` - this is only needed for the initial build
  2. `gulp watch-local`
  3. To use the changes in the developer VS Code instance:
     - Press CTRL+SHIFT+P
     - Type "Restart TS server"

3. Testing

This section describes how to write and run tests.

3.1. Write tests

- Testfiles are located in */TypeScript/tests/cases/fourslash/*
- Naming convention is as follows: `refactor<refactor_name>_<testcase_name>.ts`
- Test-file structure:
  - After `////` comes the initial state
  - `/*a*/` is a marker for the selection
  - `newContent` contains the expected code state after refactoring

```
/// <reference path='fourslash.ts' />

//// function /*a*/catTheGreat()/*b*/: void { };

goTo.select("a", "b");
edit.applyRefactor({
    refactorName: "Kitty enchantment",
    actionName: "Invoke kitty",
    actionDescription: "Invoke kitty",
    newContent: `function catTheGreat(): string {
    return "Meow";
};`,
});
```

3.2. Run tests

Tests can either be run with jake or with gulp.

- With jake:
  - Run all tests: `jake runtests-parallel`
  - Run selected tests: `jake runtests tests=<regex>`
- With gulp:
  - Run all tests: `gulp runtests-parallel`
  - Run selected tests: `gulp runtests --tests=<regex>`
  - It is much faster without linting: `gulp runtests --tests=<regex> --lint=false`
- Debugging test (to our knowledge, breakpoints do not work in production code, only in test code):
  1. To add a breakpoint, put "debugger;" in code.
  2. Run test as `gulp runtests --tests=<regex> --inspect`
  3. Open Chrome and open the location "chrome://inspect"

4. Handling certain TSLint messages

- Message: “Tag argument with parameter name”:
  - Before: `createBlock([returnStatement], true);`
  - Fix: `createBlock([returnStatement], /*multiLine */true);`
- Message: “’foo’ is declared but its value is never read.”

  To ignore an unused parameter, prepend an underscore:
  - Before: `values.map((foo, item) => item.ToUpperCase());`
  - Fix: `values.map((_foo, item) => item.ToUpperCase());`