C++ Concept Refactorings

Jeremy Stucki
Vina Zahnd

Advisor:
Thomas Corbat

21.12.2023

Figure 1: Artistic interpretation of this project [1]

OST
Eastern Switzerland
University of Applied Sciences

Abstract

With C++20, template parameter constraints [2] were introduced that allow to specify the expected functionality of the template parameters. The objective of this project was to find novel refactoring operations related to concepts and to implement them as part of a language server.

An analysis has been performed on the language server protocol, concepts, and the clangd language server. This resulted in a few potential refactoring operations, two of which have been implemented as a tweak in the clangd language server.

The first implemented refactoring enables inlining *requires* clauses into the template declaration. It reduces the amount of code and, in most cases, makes the function signature easier to read.

The second implemented refactoring allows converting explicit template declarations into their abbreviated form using *auto* parameters, thus eliminating the template header above the function.

The two implemented refactor operations have been submitted upstream as pull requests to the LLVM repository and, as of the writing of this paper (December 2023), are awaiting review. Once approved and merged, these new refactoring operations will become available to anyone using the clangd language server.

Management Summary

The goal of this project was to add new refactorings to the clangd language server to support the use of concepts, which were introduced with C++20.

Two new refactoring operations were implemented and the resulting patches have been submitted to the LLVM project. As of December 2023, the pull requests opened to merge the implemented refactorings into the LLVM project are awaiting review.

Inline Concept Requirement Inlines type requirements from *requires* clauses into the template definition, eliminating the *requires* clause. An example of its capabilities is shown in Table 1.

Abbreviate Function Template Eliminates the template declaration by using `auto` parameters. An example of its capabilities is shown in Table 2.

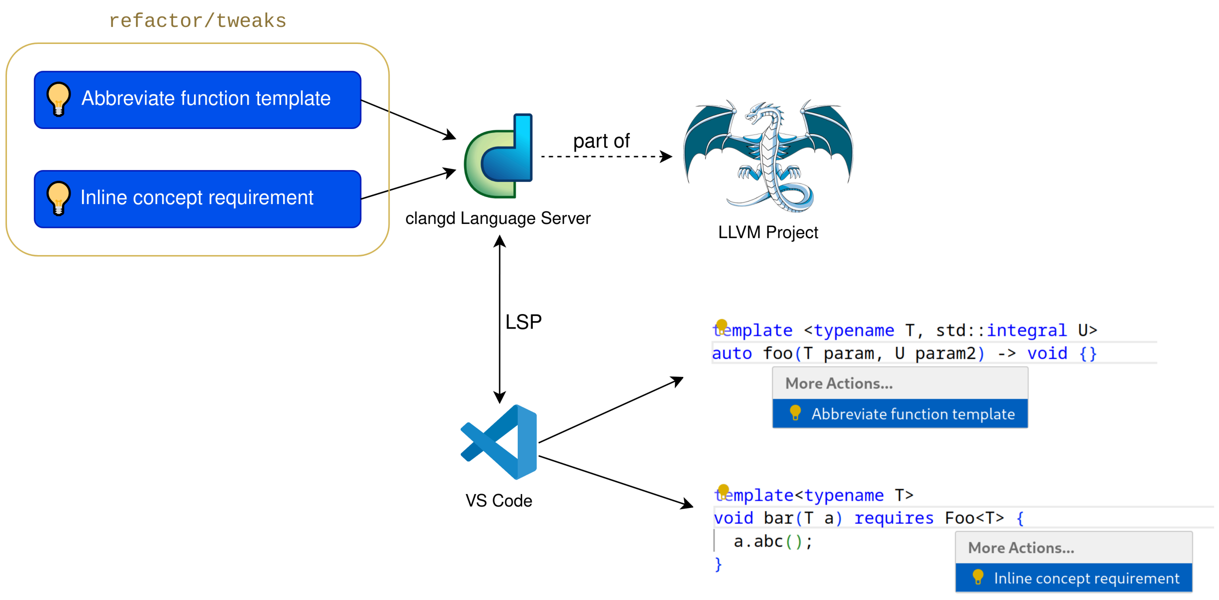
The refactoring operations were implemented as part of the clangd language server. Figure 2 shows a diagram of how VS Code is using the clangd language server to display refactoring operations. It is communicating with the language server using the language server protocol, for which the “clangd” extension can be used.

Before	After
`template <typename T>` `void foo(T) requires std::integral<T> {}`	`template <std::integral T>` `void foo() {}`

Table 1: Example of “Inline Concept Requirement” refactoring

Before	After
`template <std::integral T>` `void foo(T param) {}`	`void foo(std::integral auto param) {}`

Table 2: Example of “Abbreviate Function Template” refactoring


refactor/tweaks
Abbreviate function template
Inline concept requirement
part of
clangd Language Server
LLVM Project
LSP
VS Code
More Actions...
Abbreviate function template
More Actions...
Inline concept requirement

```
template <typename T, std::integral U>
auto foo(T param, U param2) -> void {}
```

```
template<typename T>
void bar(T a) requires Foo<T> {
  a.abc();
}
```

Figure 2: Diagram showing integration of implemented refactoring

Key Findings

- The clangd documentation is well-written and provides good support.
- Parts of the code within the LLVM project are quite old and use older language features.
- Pull requests often take a significant amount of time for reviewers to approve or even review.
- Clangd contains functions which were irritating and hard to understand and therefore leading to wrong conclusions.

Critical Issues and Challenges

- Building clangd for the first time takes a lot of cpu time and memory. This resulted in initial builds taking multiple hours.
- Finding out how to add reviewers to the pull requests posed a considerable challenge due to the absence of instructions. It appeared that the automated system malfunctioned, failing to allocate reviewers as intended.

Conclusions

Language servers offer an effective method to provide language support across multiple IDEs. The presence of an open-source project such as LLVM is not only a commendable initiative, but also receives widespread appreciation among developers in the community. Conversely, this circumstance contributes to a slower integration of new changes, given that a majority of contributors are engaged in the project during their leisure hours, impacting the pace of development.

One of the pull requests got a review from fellow contributor, who expressed anticipation for the integration of the refactoring in clangd, highlighting its potential usefulness. Their comment serves as a promising conclusion to the project's development, and it is hoped that others will similarly perceive this addition as beneficial to the language server.

Table of Contents

Abstract i
Management Summary ii
1. Introduction 1
  1.1. Initial Situation 1
  1.2. Problem Description 2
  1.3. Project Goal 2
  1.4. Structure of This Report 3
2. Analysis 4
  2.1. Refactoring 4
  2.2. Language Server Protocol (LSP) 6
  2.3. LLVM Project 9
  2.4. The clangd Language Server 9
  2.5. Refactorings in clangd 10
  2.6. Abstract Syntax Tree (AST) 12
  2.7. C++ Concepts 15
3. Refactoring Ideas 16
  3.1. Requirement Transformation 16
  3.2. Extraction of Conjunctions and Disjunctions 17
  3.3. Concept Generation 18
4. Refactoring — Inline Concept Requirement 19
  4.1. Analysis 20
  4.2. Implementation 23
  4.3. Limitations 24
  4.4. Usage 26
5. Refactoring — Abbreviate Function Template 27
  5.1. Analysis 28
  5.2. Implementation 31
  5.3. Limitations 32
  5.4. Usage 33
6. Development Process 34
  6.1. Workflow 34
  6.2. Setup 35
7. Project Management 37
  7.1. Approach 37
  7.2. Project Plan 38
  7.3. Time Tracking 39
8. Conclusion 41
  8.1. Learnings 41
  8.2. Outlook 42
9. Disclaimer 43
10. Glossary 44
11. Bibliography 45
12. Table of Figures 48
13. Table of Tables 49
14. List of Listings 50
15. Appendix 51
  15.1. Personal Report — Jeremy Stucki 52

15.2. Personal Report — Vina Zahnd 53
15.3. Assignment 54
15.4. Source Code 57

1. Introduction

Writing clean and readable code is getting more important as programming languages are growing and evolving. This is also the case for C++, therefore, refactoring already written code is becoming more important.

> Aside from the problems that could affect any language, C++ developers find code refactoring more challenging, in part because the language is complex. This complexity is compounded with language-extending functionality such as macros and templates, not to mention that the language is large, to begin with, and the syntax is difficult to process.
>
> — Dori Exterman [3]

The task of the project involves implementing and contributing new refactoring features to the LLVM project in order to assist the C++ community with their refactoring tasks.

1.1. Initial Situation

The C++ programming language is constantly developed further by the standard committee [4]. With C++20, template parameter constraints were introduced which allow specification of requirements for template arguments. Concepts are a way of abstracting these constraints. [5]

Refactoring is a common technique to resolve code smells and improve the internal structure of code without changing its external behavior. [6] Automated tests often ensure that the correct functionality is retained.

Older versions of integrated development environments (IDEs) were implementing support for code analysis and tools like symbol lookup and refactorings themselves. This led to the problem that new languages only slowly gained adoption, one editor at a time, each of them having to spend the effort to implement support for it. The goal of the Language Server Protocol was to address this and have the compiler or an adjacent tool implement the logic of these IDE features independently of a specific editor in something called a Language Server. Editors then only need to know how to communicate with this Server and they gain support for a wide range of languages. [7]

The new constructs of C++20 concepts provide the potential to apply established refactorings, and there is also the possibility of developing new refactorings. The LLVM project [8] is an open source project, whose source code is available on GitHub. It contains the source code for LLVM, a toolkit for the construction of highly optimized compilers, optimizers, and run-time environments. Clangd is a language server which lives within the LLVM project. It is able to recognize and make use of C++ code and contains smart features like code completion, compile errors and go-to-definitions.

1.2. Problem Description

When developing in any programming language features like code refactorings are a helpful tool. They can help to restructure and simplify source code. To make these features available to as many IDEs as possible the language server protocol can be used.

One language server for C++ is clangd, which unfortunately does not have many refactorings available, especially not for features introduced with C++20. Therefore, it would be nice to have some support for new language features like concepts. It would make development much more convenient and make the developer aware of other ways of writing code using the newly added features.

1.3. Project Goal

This section describes the goals of this project according to the task assignment *Section 15.3, Assignment*. Additionally, parts were added to give the project more structure, as this project is more explorative than usual.

The goal of this semester thesis is to come up with new ideas for refactoring operations specific to parameter type constraints and to implement some of them. It should be checked if currently existing refactorings can be applied to concepts. This may already be implemented in the currently available tooling. Ideally, new refactorings should be submitted upstream as a pull request to clangd. This is done to support the C++ community as well as helping the LLVM project grow.

In addition to this, research will also be carried out to determine how the clangd language server is communicating with the development tools. This also includes documenting the basic knowledge needed to understand it.

For the implementation itself, it needs to be clear where the code needs to be added, how it should be tested, what the coding guidelines are, and how it can be contributed. Each implemented refactoring feature should be documented, including is usage and how it transforms the source code.

1.4. Structure of This Report

This report encompasses the analysis, elaboration, and implementation of the project's work. It is structured into the following sections:

***Section 2, Analysis*:** Captures the findings from the research conducted on the foundational principles of the language server protocol and clangd in particular.

***Section 3, Refactoring Ideas*:** Lists the collected ideas for potential refactorings.

***Section 4, Refactoring — Inline Concept Requirement*:** Describes the implementation process and result of the refactoring "Inline Concept Requirement".

***Section 5, Refactoring — Abbreviate Function Template*:** Describes the implementation process and result of the refactoring "Convert Function Template to Abbreviated Form".

***Section 6, Development Process*:** Gives insight about how the development environment was set up and which steps were needed to make the LLVM project compile locally.

***Section 7, Project Management*:** Outlines how the project was approached and explains the project plan and time tracking.

***Section 8, Conclusion*:** Summarizes key findings, insights, and implication of the project.

2. Analysis

This section documents the research and analysis of various processes and constructs. First, in *Section 2.1, Refactoring*, it is explored what a refactoring is. Then the language server protocol is described in *Section 2.2, Language Server Protocol (LSP).*

Section 2.3, LLVM Project analyzes the structure of the LLVM project. The clangd language server is looked at in *Section 2.4, The clangd Language Server*, with a focus on refactorings, including their testing in *Section 2.5, Refactorings in clangd.*

In *Section 2.6, Abstract Syntax Tree (AST)*, abstract syntax trees are looked at, including their role in compilers and clang specifically.

Finally, C++ concepts are examined in *Section 2.7, C++ Concepts.*

2.1. Refactoring

When applying a refactoring, the external behavior needs to stay the same as before the refactoring was applied [9]. To ensure that a refactoring does not affect the behavior and does not introduce new bugs, tests should be written before a refactoring is applied. Refactoring code is a very important step while developing to improve code readability and reduce complexities. [6]

Many IDEs offer refactoring features to help the developers keep their code in good shape. To offer refactorings, the provided feature needs to be tested well to ensure that the external behavior stays the same. In a lot of cases, automated tests are used to do so. Even if all tests succeed, one would theoretically still be required to prove that the expected result has the same behavior as the test input, however, due to these tests being very concise, their correctness can typically be verified through a quick inspection. The testing of refactorings in clangd is explored in more detail in *Section 2.5.1, Testing*.

Different refactorings vary in complexity. For example, renaming a local variable typically has limited potential for unexpected side-effects. In most cases, it is sufficient to check whether the new name already exists in the affected scope. The "Abbreviate Function Template" refactoring (*Section 5, Refactoring — Abbreviate Function Template*), on the other hand, could have surprising side-effects, in which case it must not be applied. It should also be considered that in some cases, not the whole code can be analyzed or is not even available.

The preprocessor capabilites of C++ [10] pose another challenge to refactoring operations. The preprocessor allows almost any name to be redefined, which can significantly complicate the refactoring process. This means that a name in the code might not correspond to its final interpretation during compilation, making it difficult to predict the impact of a refactoring. For instance, a seemingly harmless rename of a variable or function could inadvertently clash with a name defined by the preprocessor, leading to unexpected behavior or compilation errors. To address this would require a thorough analysis and understanding of the entire codebase, including preprocessor directives, to ensure a safe refactoring.

In Table 3 an example of bad refactoring is shown. A function is defined with the template type parameters `T` and `U` and the function parameters `p1` and `p2`, which use the template type parameters in reverse order. If a refactoring, for example, converts the functions to their abbreviated form using `auto` parameters and blindly uses `auto` for all function parameter types, it would result in a different function signature. The compiler may throw an error at the call-site because the function call is no longer valid, but the updated source code could also just compile if the parameter types are all inferred. In fact, this change should not be considered a refactoring, as the external behavior changed. Therefore, it is more accurately described as a code transformation.

	Identical Parameter Types	**Different Parameter Types**	**Standard Usage**
	`f<int, int>` `(24, 42);`	`f<string, int>` `(42, "?");`	`f(42, "?");`
`// Before Refactoring` `template <typename T, std::integral U>` `void f(U p1, T p2)` `{}`	Compiles	Compiles	Compiles
`// After Refactoring` `void f(std::integral auto p1, auto p2)` `{}`	Compiles	Not Compiling Template arguments do no longer fulfill the concept requirement.	Compiles Can throw errors depending on the function body.

Table 3: A bad example of refactoring

2.2. Language Server Protocol (LSP)

The language server protocol, short LSP, is an open, JSON-RPC based protocol designed to communicate between code editors or integrated development environments (IDEs) and language servers. It provides language-specific features such as code completion, syntax highlighting, error checking, and other services to enhance the capabilities of code editors. Traditionally, this work was done by each development tool as each provides different APIs for implementing the same features.

Figure 3 shows an example for how a tool and a language server communicate during a routine editing session.

The development tool sends notifications and requests to the language server. The language server can then respond with the document URI and position of the symbol's definition inside the document for example.

By using a common protocol the same language server can be used by different editors which support the protocol. This reduces the effort required to integrate language-specific features into various development environments, allowing developers to have a more efficient and feature-rich coding experience, regardless of the programming language they are working with.

The idea of the LSP as described by Microsoft:

> The idea behind the Language Server Protocol (LSP) is to standardize the protocol for how such servers and development tools communicate. This way, a single Language Server can be re-used in multiple development tools, which in turn can support multiple languages with minimal effort.
>
> — Microsoft

Language servers are used within modern IDEs and code editors such as Visual Studio Code, Atom and Sublime Text.

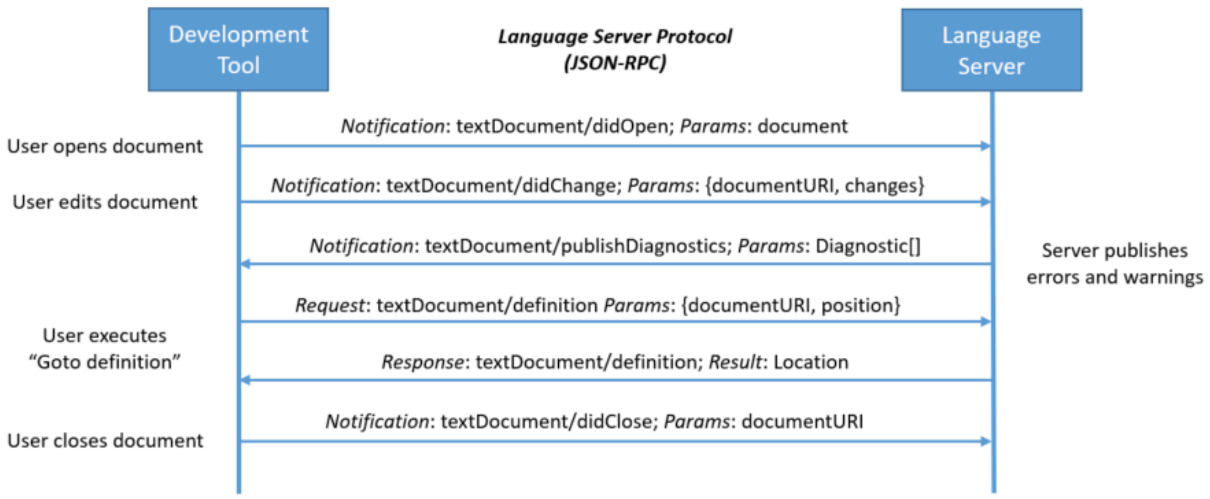

Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
Request: textDocument/definition Params: {documentURI, position}
User executes "Goto definition"
Response: textDocument/definition; Result: Location
Notification: textDocument/didClose; Params: documentURI
User closes document

Figure 3: Diagram showing example communication between IDE and language server [12]

Implementations The language servers implementing the LSP for C++ are shown in Table 4. For this project, the focus is set on the LLVM project, which is explored in *Section 2.3, LLVM Project*.

A list of tools supporting the LSP can be found on the official website [13].

Language	Maintainer	Repository	Implementation Language
C++	Microsoft	VS Code C++ extension	C++
C++/clang	LLVM Project	clangd	C++
C/C++/Objective-C	Jacob Dufault, MaskRay, topisani	cquery	C++
C/C++/Objective-C	Jacob Dufault, MaskRay, topisani	MaskRay	C++

Table 4: C++ language servers implementing the LSP [14]

2.2.1. LSP Features for Refactoring

To apply a refactoring using the LSP three steps are needed. These steps are the same for all tools using the LSP.

1. Action Request
2. Execute Command
3. Apply Edit

The flow chart Figure 4 shows a quick overview of the requests used for refactoring features. The details of the requests shown in the flow diagram are explained further in the following sections.

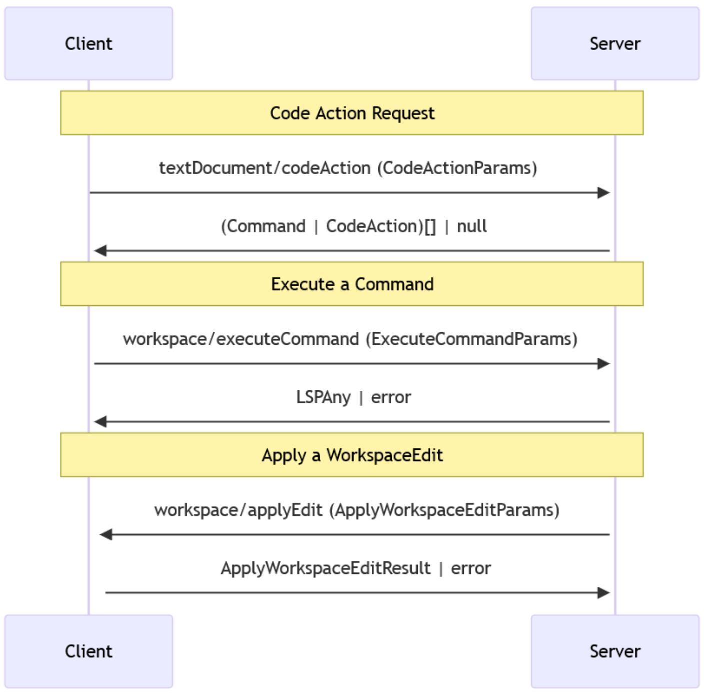

Client
Server
Code Action Request
textDocument/codeAction (CodeActionParams)
(Command | CodeAction)[] | null
Execute a Command
workspace/executeCommand (ExecuteCommandParams)
LSPAny | error
Apply a WorkspaceEdit
workspace/applyEdit (ApplyWorkspaceEditParams)
ApplyWorkspaceEditResult | error
Client
Server

Figure 4: Diagram showing code action and code action resolve request

Code Action Request The code action request is sent from client to server to compute commands for a given text document and range. To make the server useful in many clients, the command actions should be handled by the server and not by the client.

A client first needs to request possible code actions. The server then computes which ones apply and sends them back in a JSON-encoded response. Listing 1 shows an example answer to the Code Action Request.

Executing a Command To apply a code change the client sends a `workspace/executeCommand` to the server. The server can then create one or multiple workspace edit structures [15] and apply the changes to the workspace by sending a `workspace/applyEdit` [16] command to the client.

Apply a WorkspaceEdit The `workspace/applyEdit` command is sent from the server to the client to modify a resource on the client side.

When the client then applies the provided changes and reports back to the server whether the edit has been successfully applied or not.

```json
[
  {
    "command": {
      "arguments": [
        {
          "file": "<path to class where request was sent>",
          "selection": {
            "end": {
              "character": 21,
              "line": 16
            },
            "start": {
              "character": 21,
              "line": 16
            }
          },
          "tweakID": "RefactorExample"
        }
      ],
      "command": "clangd.applyTweak",
      "title": "A Refactoring Example"
    },
    "kind": "refactor",
    "title": "A Refactoring Example"
  }
]
```

Listing 1: Example answer to a code action request

2.3. LLVM Project

The LLVM project [8] is a collection of modular and reusable compiler and toolchain technologies. One of the primary sub-projects is Clang which is a "LLVM native" C/C++/Objective-C compiler.

Figure 5 illustrates how different compilers can use the LLVM intermediate language as an intermediate step before being compiled to platform-specific code.

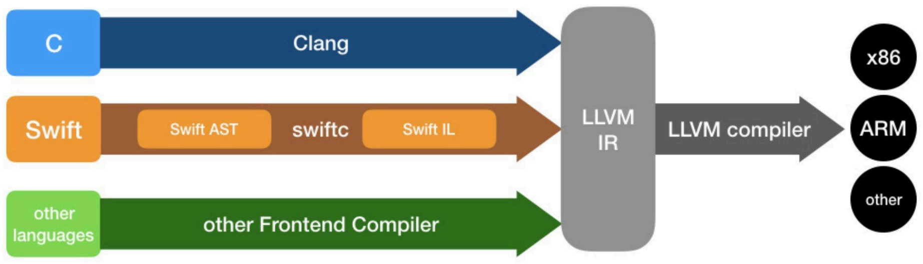


Figure 5: Diagram showing the architecture of LLVM [17]

Code refactorings for C++ can be found within the clangd language server which is based on the clang compiler. [18]

Coding Guidelines As all big projects LLVM also has defined coding guidelines [19] which should be followed. The documentation is well-written and is easy to understand which makes it easy to follow. A lot of guidelines are described, however, some things seem to be missing, like the usage of trailing return types introduced with C++ 11 [20]. When code is submitted upstream, Clang-Tidy [21] is running automatically and needs to succeed for the code to be accepted. Clang-Tidy is an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis. [21] For any other guidelines, there is no automated checking.

Code Formatter To fulfill the formatting guidelines there is a formatter `clang-format` [22] within the project to format the files according to the guidelines. A check run on GitHub is ensuring that the format of the code is correct. Only when the formatter has been run successfully a pull request is allowed to be merged.

2.4. The clangd Language Server

Clangd is the language server which lives in the LLVM project repository [8] under `clang-tools-extra/clangd`. It understands C++ code and provides smart features like code completion, compiler warnings and errors, as well as go-to-definition and find-references capabilities. The C++ refactoring features can be found within clangd under the `tweaks` folder.

2.5. Refactorings in clangd

The LLVM project is quite big and it took a while to figure out how it is structured. For refactoring features, classes can be created in `llvm-project/clang-tools-extra/clangd/refactor/tweaks`. Each refactoring operation within clangd is implemented as a class, which inherits from a common `Tweak` ancestor. To understand how a refactoring is implemented it helps to look at already existing code. Further information about how this class is used can be found in *Section 2.5.2, Code Actions*.

As concepts were introduced with C++20, it is quite new to the world of C++. Looking at the existing refactoring operations, only little support for template parameter constraints is provided. The only tweaks which can be applied are “Rename”, “Dump AST” and “Annotate highlighting tokens” (hidden by default). All other refactorings in clangd can not be applied to template parameter constraints. Some basic ones like symbol rename already work for them.

2.5.1. Testing

The LLVM project strictly adheres to a well-defined architecture for testing. To align with project guidelines [23], automated unit tests must be authored prior to the acceptance of any code contributions. The name of these files is usually the name of the class itself and uses the googletest framework [24].

Unit tests for tweaks are added to `llvm-project/clang-tools-extra/clangd/unittests`. To test them three functions are typically used, `EXPECT_EQ`, `EXPECT_AVAILABLE`, and `EXPECT_UNAVAILABLE`.

`EXPECT_EQ` Executes the `apply` function for a given snippet at a given cursor location and compares the result with the expected code.

`EXPECT_AVAILABLE` Checks if the refactoring feature is available on certain cursor position within the code. The `prepare` function is executed.

`EXPECT_UNAVAILABLE` Checks if the refactoring feature is unavailable on certain cursor position within the code. The `prepare` function is executed.

2.5.2. Code Actions

All refactoring features, so-called “tweaks”, reside in the `refactor/tweaks` directory, where they are registered using the `REGISTER_TWEAK` macro. These compact plugins inherit the tweak class [25], which acts as an interface base (Figure 6). The structure of this class is demonstrated in Listing 2. When presented with an AST and a selection, they can swiftly assess if they are applicable and, if necessary, create the edits, potentially at a slower pace. These fundamental components constitute the foundation of the LSP code-actions flow.

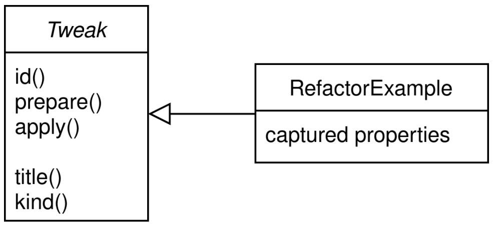


Figure 6: Class digram for clangd tweak and refactor example

```cpp
namespace clang {
namespace clangd {
namespace {
/// Feature description
class RefactorExample : public Tweak {
public:
  const char *id() const final;

  bool prepare(const Selection &Inputs) override;
  Expected<Effect> apply(const Selection &Inputs) override;
  std::string title() const override { return "A Refactoring Example"; }
  llvm::StringLiteral kind() const override {
    return CodeAction::REFACTOR_KIND;
  }
};

REGISTER_TWEAK(RefactorExample)

bool RefactorExample::prepare(const Selection &Inputs) {
  // Check if refactoring is possible
}

Expected<Tweak::Effect> RefactorExample::apply(const Selection &Inputs) {
  // Refactoring code
}

} // namespace
} // namespace clangd
} // namespace clang
```

Listing 2: Structure of a tweak class

`bool prepare(const Selection &Inputs):`
Within the `prepare` function a check is performed to see if a refactoring is possible on the selected area. The function is returning a boolean indicating whether the action is available and should be shown to the user. As this function should be fast only non-trivial work should be done within. If the action requires non-trivial work it should be moved to the `apply` function. During this phase a refactoring can also set member variables that `apply` is going to use afterwards.

For example, in VS Code the function is triggered as soon as the "Refactoring" option is used. However, LSP clients can choose to call the prepare function whenever they want.

`Expected<Tweak::Effect> apply(const Selection &Inputs):`
Within the `apply` function the actual refactoring is taking place. The function is triggered as soon as the refactoring tweak has ben selected. It is guaranteed that the `prepare` function has been called before, so the member variables it prepared can be used. It returns the edits which should be applied on the client side.

2.6. Abstract Syntax Tree (AST)

The Abstract Syntax Tree, short AST, is a syntax tree representing the abstract syntactic structure of a source code. It represents the syntactic structure of source code written in a programming language, capturing its grammar and organization in a tree data structure. It provides a structured way to analyze code, making it easier for tools like code analyzers, code editors, and IDEs to understand and work with the code. [26]

The tree underpinning the AST is a structure consisting of a root node, which is the starting node on top of the tree, which then points to other values and these values to others as well. Figure 7 shows a simple tree.

Each circle is representing a value which is referred to as a ‘node’. The relationship within the tree can be described by using names like ‘parent node’, ‘child node’, ‘sibling node’ and so on. There is no common AST representation and the structure of it varies depending on the language and compiler. [26]

To illustrate how source code gets mapped to an AST we can look at an example. Listing 3 shows a simple function that calculates the factorial of n along with a possible AST for it.

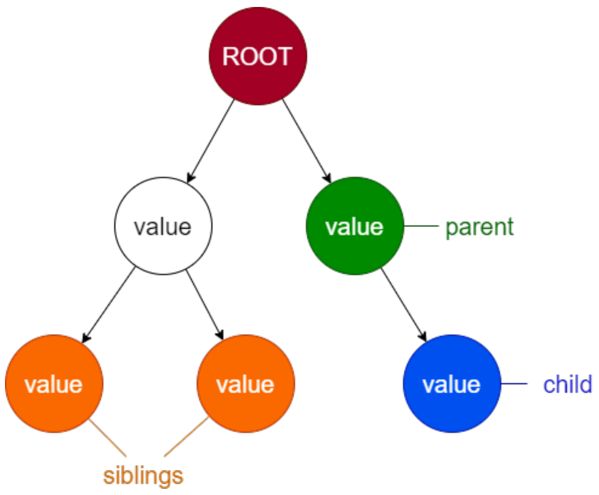
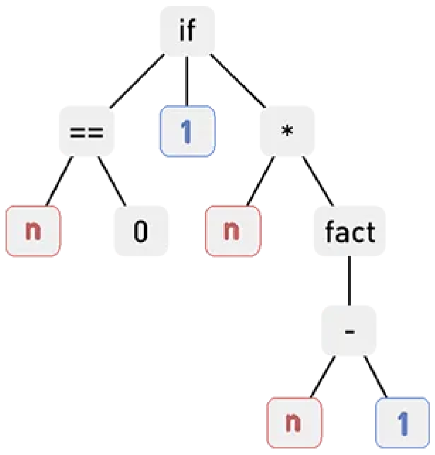


Figure 7: Diagram showing tree structure

```
int fact(int n) {
  if (n == 0)
      return 1;
  else
      return n*fact(n-1);
}
```



Listing 3: Function for calculating the factorial of a number

The most common use for ASTs are with compilers. For a compiler to transform source code into compiled code, three steps are needed. [27] Figure 8 shows a visualization of the conversion steps.

1. **Lexical Analysis** - Convert code into set of tokens describing the different parts of the code.
2. **Syntax Analysis** - Convert tokens into a tree that represents the actual structure of the code.
3. **Code Generation** - The compiler can apply multiple optimizations and then convert it into binary code.

In Clangd, the "AST" refers to the AST generated by the Clang compiler for a C++ source file. Clangd uses this AST to provide features like code completion, code navigation, and code analysis in C++ development environments.

Clang also has a builtin AST-dump mode, which can be enabled with the `-ast-dump` flag. [28] Listing 4 shows an example of the ast dump of a simple function. All Clang AST nodes are described in the generated online documentation Doxygen [29].

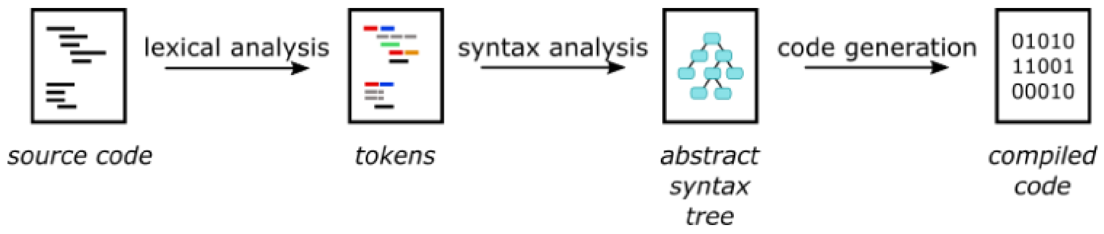


Figure 8: Diagram showing steps for code generation [27]

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

Listing 4: Example of AST dump in clang [28]

When building an AST a root node needs to be found. In clang, the top level declaration in a translation unit is always the `translation unit declaration` [30]. For a function for example it would be the `function declaration` [31]. When the code has errors the AST can only be built if the top level declaration is found. For a function this means, that the function name has to be present but other than that it can have errors within the code.

Table 5 shows in which cases the AST can or can not be built.

`int f(int x) {`
`  int result = (x / 42);`
`  return result;`
`}`
	OK

`f(int x) {`
`  int result = (x / 42);`
`  return result;`
`}`
	OK

`int f(int x) {`
`  int result = (x / 42);`
`}`
	OK

`int f {`
`  int result = (x / 42);`
`  return result;`
`}`
	OK

`int (int x) {`
`  int result = (x / 42);`
`  return result;`
`}`
	NOT OK
	function name missing

`int f(int x)`
`  int result = (x / 42);`
`  return result;`
	NOT OK
	function brackets missing

Table 5: AST dump possibilities for valid and invalid functions

2.7. C++ Concepts

Concepts are a new language feature introduced with C++20. They allow puttign constraints on template parameters, which are evaluated at compile time. This allows developers to restrict template parameters in a new convenient way. For this the keywords `requires` [32] and `concept` [33] were added to give some language support. [34]

Before C++20 `std::enable_if` [35] was used for such restrictions, more about these can be found in the C++ stories [36].

Usage of the `requires` keyword The `requires` keyword can be used either before the function declaration or between the function declaration and the function body as illustrated in Figure 9. The requirements can also contain disjunctions (`||`) and/or conjunctions (`&&`) to restrict the parameter types even more. It should be noted that nested require clauses are possible, which allows for many additional constructs. Examples for these can be found in Listing 5 and Listing 6.

Usage of the `concept` keyword Using the `concept` keyword requirements can be named, so they do not have to be repeated for every `requires` clause. An example for a concept declaration can be found in Listing 7.

```cpp
template <typename T>
requires CONDITION
void f(T param)
{}
```

```cpp
template <typename T>
void f(T param) requires
CONDITION
{}
```

Figure 9: Functions using requires clause

```cpp
requires std::integral<T> || std::floating_point<T>
```

Listing 5: Requires clause using disjunction

```cpp
requires std::integral<T> && std::floating_point<T>
```

Listing 6: Requires clause using conjunction

```cpp
template<typename T>
concept Hashable = requires(T a)
{
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};
```

Listing 7: Hashable concept declaration

3. Refactoring Ideas

In this section ideas for potential refactoring operations are explored. This serves as the foundation for deciding which features to implement. A total of three ideas are described.

The concepts outlined in this section are intentionally presented in a basic state. The objective is to subject subsets of these concepts to further analysis and refinement for each refactoring operation during the implementation phase.

The first idea, described in *Section 3.1, Requirement Transformation*, is inspired by sample code from the constraints and concept reference [5]. The second idea, described in *Section 3.2, Extraction of Conjunctions and Disjunctions*, came up during experimentation with concepts. Finally, the third idea (*Section 3.3, Concept Generation*) describes automatic concept declaration generation.

3.1. Requirement Transformation

A refactoring could be provided to transform a function template using constraints between alternate forms. Table 6 shows different variations of a function template. They all result in an identical function signature after instantiation and thus cannot co-exist. This was verified using gcc and clangd.

The benefit of this is in many cases a more readable function declaration. For more readable code some developers prefer to remove unnecessary code like the `requires` keyword. The versions on the left in Table 6 shows how the code looks like without the keyword resulting in the same logic. The potential refactoring would therefore focus on the removal of the `requires` clause.

This idea was inspired by the constraints and concept reference [5] since it lists all these forms in its first code snippet.

```cpp
template<Hashable T>
void f(T) {}
```

```cpp
template<typename T>
void f(T) requires Hashable<T> {}
```

```cpp
void f(Hashable auto x) {}
```

```cpp
template<typename T> requires Hashable<T>
void f(T) {}
```

Table 6: Different ways to constrain a function template using concepts

3.2. Extraction of Conjunctions and Disjunctions

Sometimes more than one constraint is used in a `requires` clause. This is expressed by `||` and `&&` operators. The proposed refactoring would offer to extract these logical combinations into a new named concept.

One possible hurdle for this refactoring could be that there is no way to trigger a rename of the newly created concept. This seems to be a limitation of the language server protocol [37], [38]. The refactoring itself would still be possible, but use a generated name for the new concept, requiring the user to rename it.

To illustrate the idea, Listing 8 shows a method `bar` whose type parameter `T` is constrained by two concepts. These requirements are extracted into a new named concept in Listing 9. The name *NAME* would be the name generated by the refactoring.

```
template <typename T>
void bar(T a) requires std::integral<T> && Hashable<T> {
  ...
}
```

Listing 8: An existing conjunction

```
template<typename T>
concept NAME = std::integral<T> && Hashable<T>;

template <typename T>
void bar(T a) requires NAME<T> {
  ...
}
```

Listing 9: The proposed refactoring to the conjunction in Listing 8

3.3. Concept Generation

A refactoring that generates a new concept declaration based on an existing function template declaration would be really useful. It would allow developers to write comprehensive concepts more easily without having to analyze their code in detail.

For example the function template in Listing 10 would result in a concept and function template declaration like the one in Listing 11 when applied to the template parameter `T`.

```cpp
template <typename T>
void foo(T a) {
  double x = a.abc();
}
```

Listing 10: Simple function template declaration

```cpp
template<typename T>
concept HasAbcMethod = requires(T a) {
  { a.abc() } -> std::convertible_to<double>;
};

template <HasAbcMethod T>
void foo(T a) {
  double x = a.abc();
}
```

Listing 11: Listing 10 with the concept extracted

4. Refactoring — Inline Concept Requirement

For this refactoring a subset of the initial idea (*Section 3.1, Requirement Transformation*) is implemented. Specifically the inlining of an explicit `requires` clause into a constrained function template. Table 7 shows some examples of what this refactoring is able to do.

A detailed analysis can be found in *Section 4.1, Analysis*. Implementation details are discussed in *Section 4.2, Implementation* and limitations are explored in *Section 4.3, Limitations*. Finally, the usage of the refactoring is shown in *Section 4.4, Usage*.

Before	After

`template <typename T>`
`void f(T) requires foo<T>`
	`template <foo T>`
	`void f(T)`

`template <typename T>`
`requires foo<T>`
`void f(T)`
	`template <foo T>`
	`void f(T)`

`template <typename T>`
`void f() requires std::integral<T>`
	`template <std::integral T>`
	`void f()`

Table 7: Capabilities of the “Inline Concept Requirement” refactoring

4.1. Analysis

The analysis will look at which elements need to be captured (*Section 4.1.2, Captured Elements*) and how the refactoring transforms the AST (*Section 4.1.3, Abstract Syntax Tree*).

4.1.1. Preconditions

The refactoring should be as defensive as possible and only apply when it is clear that it will apply correctly. In Table 8 checks are explained which are made during the preparation phase to ensure the refactoring feature can be applied.

Check	Reasoning
The selected `requires` clause only contains **a single requirement**, e.g. `requires CONDITION`	Combined concept requirements are complex to handle and would increase the complexity drastically. This is a temporary restriction that could be lifted in the future.
The selected `requires` clause only contains **a single type argument**, e.g. `requires std::integral<T>`.	This case is complex to handle and would increase the complexity drastically. This is a temporary restriction that could be lifted in the future.
The concept requirement has a parent of either a **function**, e.g. `template<>` `void f() requires CONDITION {}` or a **function template**, e.g. `template<>` `requires CONDITION` `void f() {}`	To restrict the refactoring operation only function templates are allowed. This is a temporary restriction that could be lifted in the future.

Table 8: Checks made during the preparation phase of the "Inline Concept Requirement" refactoring

4.1.2. Captured Elements

Capturing an element means finding it in the AST and keeping a reference to it for the application phase. Figure 10 shows the captured elements and their purpose. A reference to them is stored as a member of the tweak object during the preparation phase and used during the application phase.

It is never mentioned explicitly that the AST references are guaranteed to be valid until the application phase, but the refactorings already present treat it as such, which is why the refactorings in this project also do so.

```
template <typename T>
          ^^^^^^^^^^
void f(T) requires foo<T> {}
```

Template Type Parameter Declaration
Will be updated using the concept found in the concept specialization expression below.

```
template <typename T>
void f(T) requires foo<T> {}
                   ^^^^^^
```

Concept Specialization Expression
Will be removed.

```
template <typename T>
void f(T) requires foo<T> {}
          ^^^^^^^^
```

Requires Token
Will be removed.

Figure 10: Elements captured for the “Inline Concept Requirement” refactoring

4.1.3. Abstract Syntax Tree

The AST gives a good overview over the structure of the code before and after the refactoring. In Figure 11 the ASTs of a simple template method and its corresponding refactored version are shown.

Looking at the original version (on the left) it is visible that the outermost `FunctionTemplate` contains the template type parameters, as well as the function definition. The `requires` clause is represented by a `ConceptSpecialization` with a corresponding `Concept reference`.

During the refactor operation most of the AST stays untouched, except for the concept reference (in yellow), which gets moved to the template type parameter and the concept specialization, which gets removed (in red).

After the examination, it was concluded that `ConceptSpecialization` nodes can be searched to see if the refactoring applies, and then additional analysis can be performed. This also guards against accidentally refactoring similar looking expressions that are not concept specializations. For example, `foo<T>` could also be a variable template.

Before

```
template<typename T>
void bar(T a) requires Foo<T> {
  a.abc();
}
```

After

```
template<Foo T>
void bar(T a) {
  a.abc();
}
```

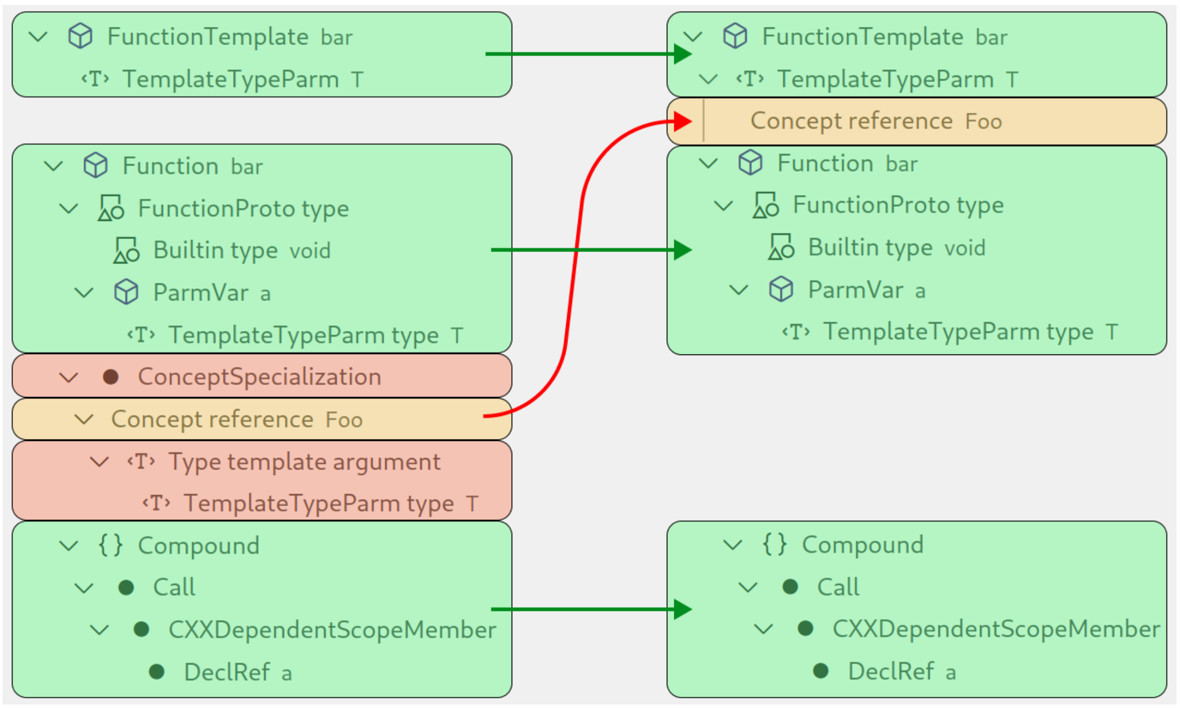


Figure 11: Example AST tranformation of the “Inline Concept Requirement” refactoring

4.2. Implementation

The implementation process was relatively straightforward, particularly after determining how to traverse the AST. The traversal just requires the current selection, which provides a helper method to get the common ancestor in the AST, after which the tree can be traversed upwards using the `Parent` property of each node and checking if the type of the node reached is the correct one.

However, there were challenges in discovering certain methods, as some were global and others necessitated casting. During this phase, referencing existing refactorings provided significant assistance.

The biggest hurdle of this refactoring was the `requires` keyword itself, which was quite hard to track down as it is not part of the AST itself. To figure out where exactly it is located in the source code it was necessary to resort to the token representation of the source range.

Testing

A lot of manual tests were performed using a test project. Debug inspections were performed often to verify assumptions. Unit tests were also written as described in *Section 2.5.1, Testing*, which consist of a total of 11 tests, 4 of them availability tests, 4 unavailability tests and 3 application tests. This is a similar extent to which existing refactorings are tested.

Pull Request

The implementation has been submitted upstream as a pull request [39] and as of December 2023 is awaiting review.

4.3. Limitations

To keep the scope of the implementation managable it was decided to leave some features out. These limitations however could be lifted in a future version. The implementation is built so it actively looks for these patterns and does not offer the refactoring operation if one is present.

4.3.1. Combined Concept Requirements

Handling combined `requires` clauses would certainly be possible, however it would increase the complexity of the refactoring code significantly. Since working on the LLVM project is new for all participants, it was decided that this feature will be left out.

```cpp
template <typename T, typename U>
requires foo<T> && foo<U>
void f(T)
```

Listing 12: Combined concept requirement

4.3.2. Class Templates

Supporting class templates would have been feasible, but a decision was made to exclude this possibility due to time constraints and in favor of maintaining simplicity in the initial refactoring implementation.

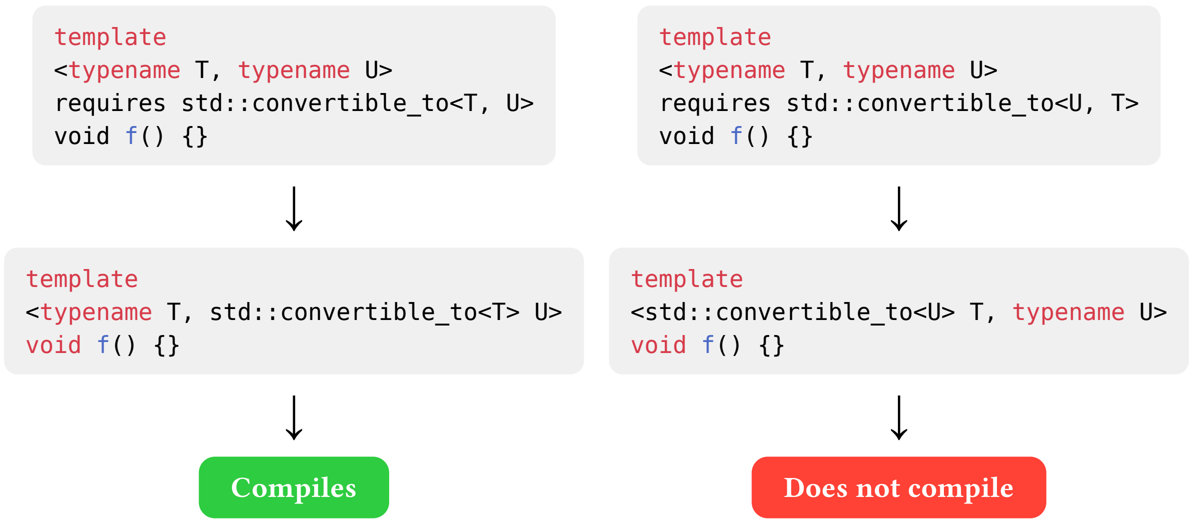
```cpp
template <typename T>
requires foo<T>
class Bar;
```

Listing 13: Class template

4.3.3. Multiple Type Arguments

If a concept has multiple type arguments, such as `std::convertible_to<T, U>` the refactoring will not be applicable. The complexity associated with managing this particular case is considerable, while the potential use case is minimal. As a result, a decision was made not to incorporate this capability.

The refactoring would be available in most scenarios involving multiple type arguments, except when the template arguments are function template parameters and are defined after the final template argument. To illustrate this, two examples are provided in Figure 12. They both show a version after the transformation has been applied. Only the version on the left represents a valid refactoring.



```
template
<typename T, typename U>
requires std::convertible_to<T, U>
void f() {}
```

```
template
<typename T, std::convertible_to<T> U>
void f() {}
```

```
template
<typename T, typename U>
requires std::convertible_to<U, T>
void f() {}
```

```
template
<std::convertible_to<U> T, typename U>
void f() {}
```

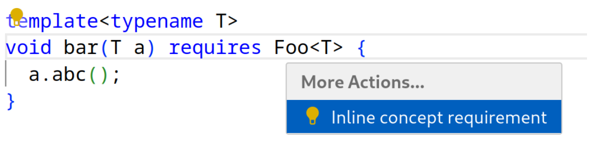
Figure 12: Example for “Inline Concept Requirement” refactoring with multiple type arguments

4.4. Usage

The refactoring is available as a code action to language server clients and is available on the whole `requires` clause.

4.4.1. VS Code

To use the feature the user needs to hover over the requires clause (e.g. `std::integral<T>`), then right click to show the code options. To see the possible refactorings the option "Refactor..." needs to be clicked and then the newly implemented refactoring "Inline concept requirement" will appear within the listed options. How this can look like is shown in Figure 13.

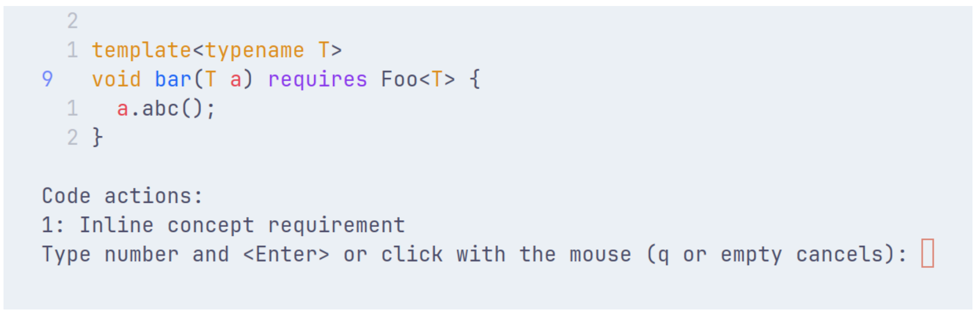


```
template<typename T>
void bar(T a) requires Foo<T> {
  a.abc();
}
```

Figure 13: Screenshot showing the option to inline a concept requirement in VS Code

4.4.2. Neovim

Figure 14 shows how the refactoring looks like before accepting it in Neovim. The cursor needs to be placed on the requires clause before triggering the listing of code actions.



```
template<typename T>
void bar(T a) requires Foo<T> {
  a.abc();
}

Code actions:
1: Inline concept requirement
Type number and <Enter> or click with the mouse (q or empty cancels):
```

Figure 14: Screenshot showing the option to inline a concept requirement in Neovim

5. Refactoring — Abbreviate Function Template

For this refactoring, another subset of the first idea (*Section 3.1, Requirement Transformation*) is implemented. It replaces explicit function template declarations with abbreviated declarations using `auto` parameters. This tweak helps reduce the number of lines and makes the code more readable. Table 9 shows examples of what this refactoring is able to do.

A detailed analysis, including call-site implications, can be found in *Section 5.1, Analysis*. Implementation details are discussed in *Section 5.2, Implementation* and limitations are explored in *Section 5.3, Limitations*. Finally, the usage of the refactoring is shown in *Section 5.4, Usage*.

Before	After

`template <typename T>`
`auto f(T param)`
	`auto f(auto param)`

`template <typename ...T>`
`auto f(T ...p)`
	`auto f(auto ...p)`

`template <std::integral T>`
`auto f(T param)`
	`auto f(std::integral auto param)`

`template <std::integral T>`
`auto f(T const ** p)`
	`auto f(std::integral auto const ** p)`

Table 9: Capabilities of the “Abbreviate Function Template” refactoring

5.1. Analysis

The analysis looks at the captured elements (*Section 5.1.2, Captured Elements*), call site implications (*Section 5.1.3, Call Site Implications*), and the impact of the refactoring on the abstract syntax tree (*Section 5.1.4, Abstract Syntax Tree*).

5.1.1. Preconditions

The refactoring should be as defensive as possible and only apply when it is clear that it will apply correctly. In Table 10 checks are explained which are made during the preparation phase to ensure the refactoring feature can be applied.

Check	**Reasoning**
A template definition needs to be in place.	If the template definition is not present the logic of this refactoring can not be applied.
The template type parameter is not used multiple times, e.g. `template<typename T>` `void f(T p1, T p2) {}`	If the type parameter is used in the body it cannot be replaced with an `auto` param.
The order of template parameters is the same as their occurrence as function parameters, e.g. `template<typename T, typename U>` `void f(T p1, U p2) {}`	The function signature would change otherwise.
The parameter type is not used within a container (e.g. `map`, `set`, `list`, `array`)	The `auto` keyword cannot be used in this context.
No `requires` clause should be present.	As the refactoring is removing the type parameter the `requires` clause would not be valid anymore.

Table 10: Checks made during the preparation phase of the "Abbreviate Function Template" refactoring

5.1.2. Captured Elements

Figure 15 shows the captured elements and their purpose. A reference to them is stored as a member of the tweak object during the preparation phase and used during the application phase.

```cpp
template <std::integral T>
^^^^^^^^^^^^^^^^^^^^^^^^^^
auto f(T param) -> void {}
```

Template Declaration
Will be removed.

```cpp
template <std::integral T>
          ^^^^^^^^^^^^^
auto f(T param) -> void {}
```

Template Parameter Restriction
Will be used in the parameter type replacement.

```cpp
template <std::integral T>
auto f(T param) -> void {}
       ^
```

Parameter Type
Will be replaced with template parameter restriction and `auto`.

Figure 15: Elements captured for the “Abbreviate Function Template” refactoring

5.1.3. Call Site Implications

The refactoring must not change the signature of the target method. In regard to this specific refactoring, the order of type parameters must stay the same. This is only the case if the `auto` parameters are in the same order as their original template parameters.

For example the two methods in Figure 16 result in two different signatures. When calling these methods with `foo<int, float>(1.0, 2)` only the version on the left would compile, as the types of the arguments would be `float` for `param1` and `int` for `param2`. The second version would be the opposite, `int` for `param1` and `float` for `param2`, which then breaks the call as the parameter types do not match.

```cpp
template<typename T, typename U>
auto foo(U param1, T param2)
```

```cpp
auto foo(auto param1, auto param2)
```

Figure 16: Example to illustrate call site differences of auto parameters

5.1.4. Abstract Syntax Tree

As can be seen in Figure 17, the AST transformation of this refactoring is very minimal. The only change is that the explicit type parameter name is replaced with a generated one.

It is interesting to see how abstract the AST really is in this case. It does not reflect the source code as closely as in the AST of the previous refactoring (*Section 4.1.3, Abstract Syntax Tree*).

The `template` is not represented as a separate node, instead its information is stored in the `TemplateTypeParam` node. Therefore, the AST structure does not change, only the information within the node is.

Before

```
template<std::integral T>
auto f(T param) -> void
{}
```

After

```
auto f(std::integral auto param) -> void
{}
```

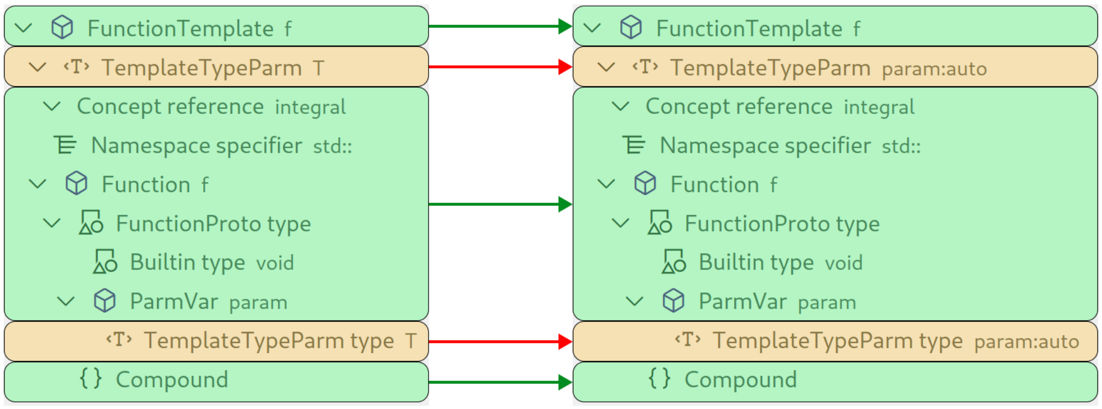


Figure 17: Example AST tranformation of “Abbreviate Function Template” refactoring

5.2. Implementation

The most challenging part of this refactoring was figuring out where template parameters are being used, as the refactoring only applies if there is exactly one usage of the parameter and that usage is as a function parameter type.

Initially, it was tried to perform a symbol lookup in the index clangd holds, but this led to no results. This is very likely due to the target being a template parameter, which has no proper symbol ID in the context of the template declaration, as it is not a concrete instance of a type.

Afterward, the way the "find references" LSP feature is implemented in clangd was analyzed. It uses a helper class called `XRefs` which implements a `findReferences` function that can deal with template functions. Unfortunately, the result of this method call cannot be traced back to the AST.

In the end, the `findReferences` call is only used to find the number of references to a given template parameter. This number is an important point of reference to see if the refactoring applies.

- If there is **only one reference**, it would mean that the template parameter is declared but never used. In this case, the refactoring cannot apply, since the template parameter would cease to exist, resulting in a different function signature.
- If there are **exactly two references**, it means that one of those is the definition and there is at least one usage of it. Figuring out if the usage is a function parameter is done in a later step.
- If there are **more than two references**, it means that there are either two function parameters with the same type or there is at least one usage outside of function parameters (for example the body of the function). In both cases the refactoring cannot apply, since replacing both usages with `auto` would result in two template parameters where there used to be just one, thus changing the function signature.

As a next step, the function parameters are iterated over to verify that each template parameter type occurs as a function parameter and that the order is the same. In addition, the type qualifiers are extracted, which consist of reference kind, constness, pointer type, and array specifiers. Function parameters are not supported for now, since their qualifiers are structured quite differently.

The application phase is rather simple in comparison. In a first step, the template declaration is removed, and in a second step, the types of the function parameters are updated. The information needed for this has been collected during the preparation phase.

Testing

A lot of manual tests were performed using a test project. Debug inspections were performed often to verify assumptions. Unit tests were also written as described in *Section 2.5.1, Testing*, which consist of a total of 14 tests, 4 of them availability tests, 4 unavailability tests and 6 application tests. This is a similar extent to which existing refactorings are tested.

Pull Request

The implementation has been submitted upstream as a pull request [40] and as of December 2023 is awaiting review.

5.3. Limitations

There are limits to this refactoring. Some of them are given by the language or compiler and some are intentional, because otherwise the scope of the refactoring would have increased drastically.

5.3.1. Templated Function Parameters

If a function parameter is a templated parameter like `std::vector<T>` the refactoring cannot apply. The reason for this is that `auto` cannot be used as a template argument.

```cpp
template <typename T>
void foo(std::vector<T> param)
```

Listing 14: Templated function parameter

5.3.2. Template Arguments Used Multiple Times

This is an inherit limitation of the refactoring. If for example the same template parameter is used for multiple function parameters, it means that all of them will have the same type when instantiated. Would they be replaced with `auto`, each of them would result in a different type.

This limitation also applies if the template argument is used anywhere else. This includes the return type and the body of the function. Replacing one template parameter with multiple `auto` keywords likely breaks the behavior of the function.

```cpp
template<std::integral T>
auto foo(T param, T anotherParam) -> void {}
```

Listing 15: Template argument used for multiple function parameters

5.3.3. Requires Clauses

Functions with `requires` clauses are not supported. As a workaround the previously implemented refactoring (*Section 4, Refactoring — Inline Concept Requirement*) can be used first.

```cpp
template <typename T>
void f(T) requires foo<T> {}
```

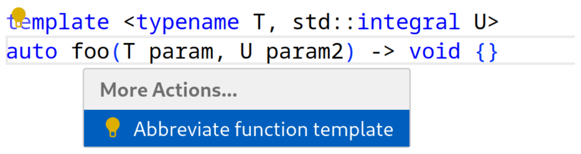
Listing 16: Function template with requires clause

5.4. Usage

The refactoring is available as a code action for language server clients. To use it the cursor can be placed anywhere within the function.

5.4.1. VS Code

To use the feature the user needs to hover over any part of the function, then right click to show the code options. To see the possible refactorings the option “Refactor...” needs to be clicked and then the newly implemented refactoring “Abbreviate Function Template” will appear within the listed options. How this can look like is shown in Figure 18.

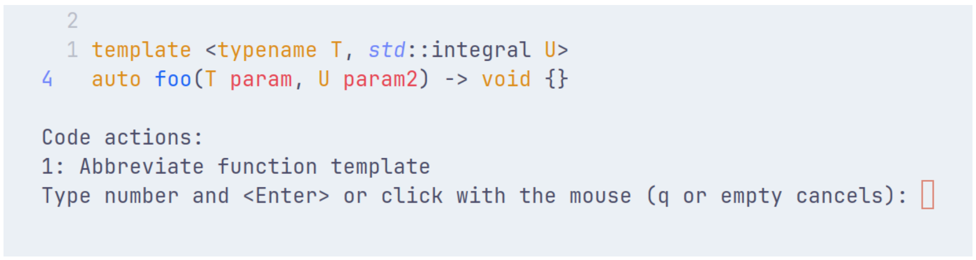


```
template <typename T, std::integral U>
auto foo(T param, U param2) -> void {}
```

Figure 18: Screenshot showing the option to abbreviate a function template in VS Code

5.4.2. Neovim

Figure 19 shows how the refactoring looks like before accepting it in Neovim. The cursor can be placed anywhere within the function before triggering the listing of code actions.



```
template <typename T, std::integral U>
auto foo(T param, U param2) -> void {}

Code actions:
1: Abbreviate function template
Type number and <Enter> or click with the mouse (q or empty cancels):
```

Figure 19: Screenshot showing the option to abbreviate a function template in Neovim

6. Development Process

Within this section, it is explained how the working environment was setup and what tools were used for development. *Section 6.1, Workflow* documents what the work process looked like. In *Section 6.2, Setup*, it is described how the project was setup for Windows and Linux.

6.1. Workflow

As the LLVM project, to which the new refactoring features should be contributed, is on GitHub, it was decided to also work using GitHub. The LLVM project already had workflows set up on GitHub, so the decision was made to re-use them. The LLVM project was then forked into a public repository [41], where for each refactoring feature, a separate branch was created. After the implementation of a refactoring feature was finished, a pull request to the LLVM project was submitted.

Two different systems were used for development (Windows and Linux), but the IDEs used were the same on both systems. To be able to use the local build of the clangd language server, a simple settings.json file can be added to VS Code. More details about this setup can be found in *Section 6.2, Setup*. For testing the local build, a test project was created containing different versions of function templates and concepts. These code snippets were then used to test the refactoring implementations using VS Code. Figure 20 illustrates how the different tools and setups work together.

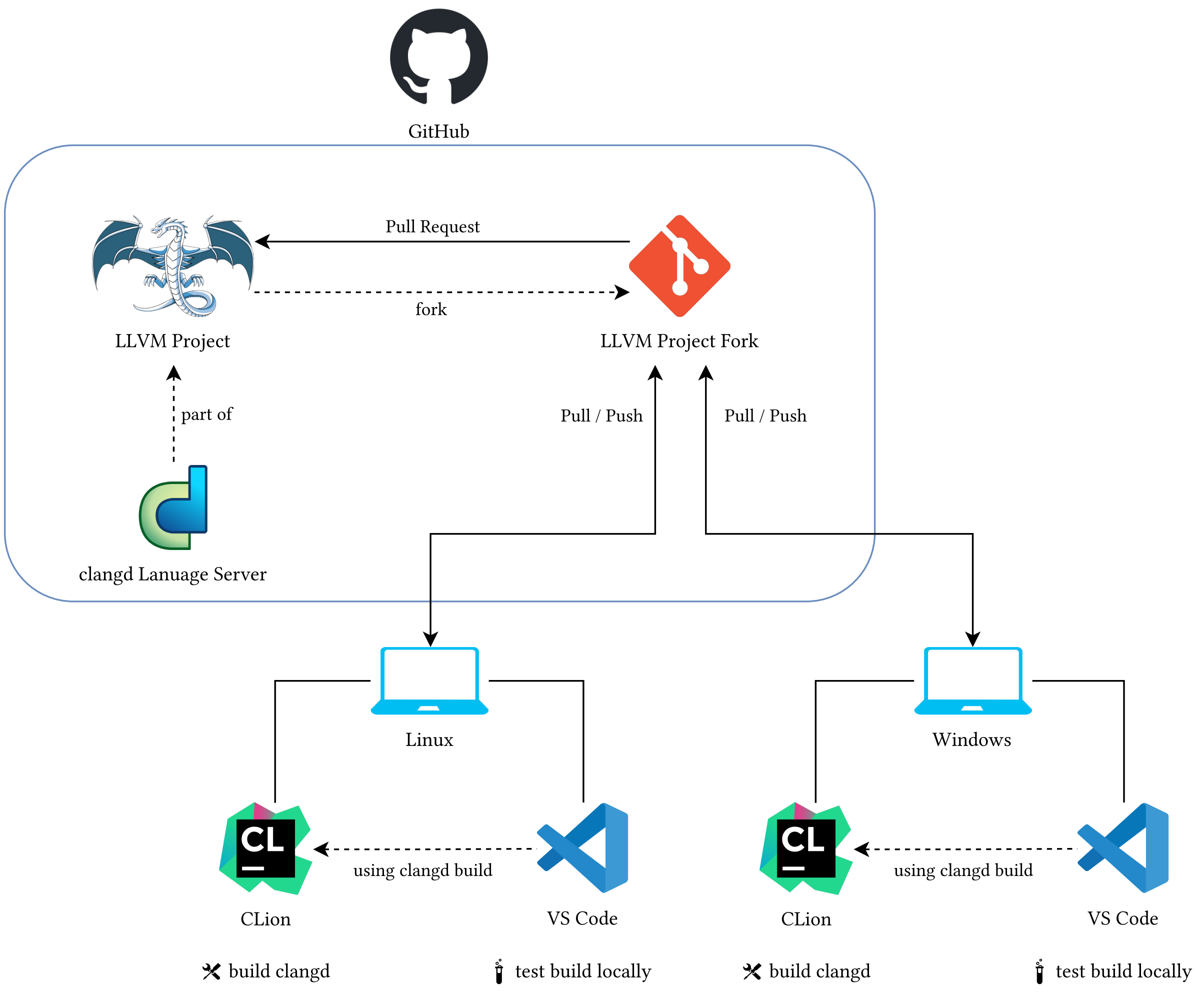


Figure 20: Diagram showing structure and workflow of the project
(Logos from [42], [43], [44], [45], [46], [47])

6.2. Setup

To build clangd, CLion was used as an IDE since it has great support for CMake as well as very good autocomplete, search, and debugging capabilities. VS Code with the clangd extension [48] and the CMake extension [49] was then configured to use the locally built language server using the `clangd.path` setting as shown in Listing 17.

```
{
    // Windows
    "clangd.path": "..\\llvm-project\\llvm\\cmake-build-release\\bin\\clangd.exe"

    // Linux
    "clangd.path": "../llvm-project/llvm/cmake-build-release/bin/clangd"
}
```

Listing 17: Configuration in settings.json file for to VS Code

For building clangd using CLion the following steps were executed.

1. Clone project from GitHub https://github.com/llvm/llvm-project
   - `git clone git@github.com:llvm/llvm-project.git`
2. Open llvm project in CLion
3. Open folder `llvm/CMake.txt` in CLion and execute cmake
4. File -> Settings -> Build, Execution, Deployment -> CMake -> CMake options
   - add `-DLLVM_ENABLE_PROJECTS='clang;clang-tools-extra'`
5. Choose “clangd” in target selector and start building[1]

After the clangd build is completed the language server within VS Code needs to be restarted to use the current build. This can be done by pressing `Ctrl + p` : `>Restart language server`.

6.2.1. Linux

The project was built using ninja and gcc12. Tests with the language server were performed using VSCodium [50], a fork of VS Code without telemetry. Neovim [51] was also used for testing, which contains a built-in LSP client. Configuration for Neovim can be found in Listing 18.

The hardware used was an AMD Ryzen™ PRO 4750U (8 core mobile) and an AMD Ryzen™ 9 5900X (12 core desktop) CPU with 48 gigabytes of system memory.

```
:lua vim.lsp.start({
  name = 'clangd',
  cmd = {'PATH_TO_CLANGD_BINARY'},
  root_dir = vim.fs.dirname(
    vim.fs.find({'CMakeLists.txt'}, { upward = true })[1]
  )
})
```

Listing 18: Configuring clangd in Neovim

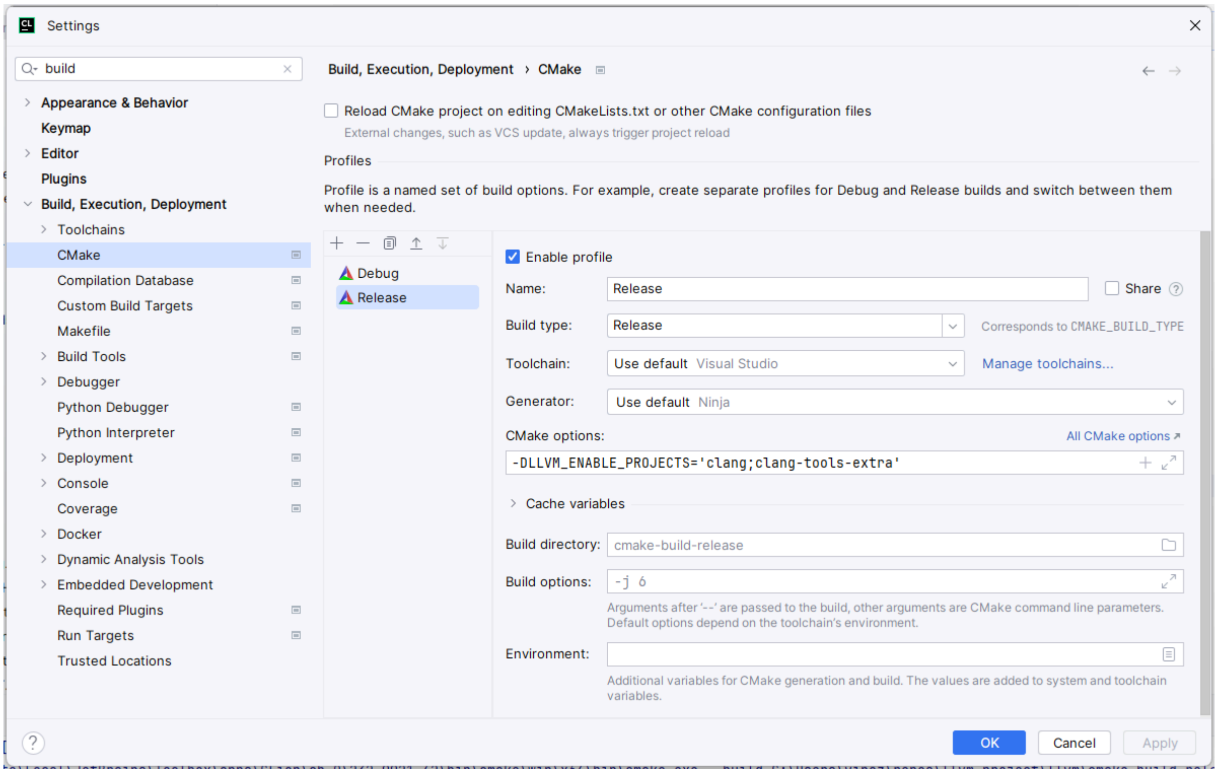
[1]When using Windows, clangd.exe should not be in use to build clangd successfully. In this example this applies to VS Code when the language server has started.

6.2.2. Windows

On Windows the project was built using ninja and Visual Studio. The hardware used was a Intel® Core™ i7-10510U CPU with 16 gigabytes of system memory.

Visual Studio was installed with the following components.

- C++ ATL for latest v143 build tools
- Security Issue Analysis
- C++ Build Insights
- Just-In-Time debugger
- C++ profiling tools
- C++ CMake tools for Windows
- Test Adapter for Boost.Test
- Test Adapter for Google Test
- Live Share
- IntelliCode
- C++ AddressSanitizer
- Windows 11 SDK
- vcpkg package manager



☐ Reload CMake project on editing CMakeLists.txt or other CMake configuration files
External changes, such as VCS update, always trigger project reload

Profiles

Profile is a named set of build options. For example, create separate profiles for Debug and Release builds and switch between them when needed.

☑ Enable profile

Name: Release ☐ Share

Build type: Release Corresponds to CMAKE_BUILD_TYPE

Toolchain: Use default Visual Studio

Generator: Use default Ninja

CMake options:

```
-DLLVM_ENABLE_PROJECTS='clang;clang-tools-extra'
```

Build directory: cmake-build-release

Build options: -j 6

Arguments after '--' are passed to the build, other arguments are CMake command line parameters. Default options depend on the toolchain's environment.

Environment:

Additional variables for CMake generation and build. The values are added to system and toolchain variables.

Figure 21: Screenshot showing build settings in CLion on Windows

7. Project Management

The approach of the project is explained in *Section 7.1, Approach*. The planning of the project is looked into in *Section 7.2, Project Plan*, including a comparative analysis between the initial plan and its effective implementation. Finally, *Section 7.3, Time Tracking* provides a summarized overview of the allocated working hours coupled with a reflective assessment of the time invested.

7.1. Approach

To discuss the ongoing process a weekly meeting with the advisor was held where it was discussed what work has been done and what is planned for the next week. Receiving a feedback of the ongoing development is quite important as the direction we are going towards can be assessed and adjusted in just a week.

The development is managed using GitHub as described in *Section 6, Development Process*. For better overview on what is already done, pull requests are used to review each others changes. This same method is also used for writing the documentation which allows tracking all changes easily.

For the documentation, Typst [52] was used, which is a new markup-based typesetting system. In VS Code, there is an extension supporting Typst [53], no other tool is needed. After the document had a stable state, pull requests were used here as well to double-check that both team members agree on what the other was writing about. GitHub issues [54] were sometimes also used to keep track of the documentations state.

The project is split into three main steps.

Project Setup Before implementing a project, a setup needs to be in place, therefore some research needed to be done to figure out what is needed to build the LLVM project and clangd specifically. The setup needed to work on Linux and Windows as both systems were used for this project.

Analysis The analysis consists of a lot of research on how the clangd language server works and how it is communicating with the IDE. This step also contains cathering all knowledge needed for implementation, such as looking into the idea of concepts and figuring out which refactoring features would be a good addition to have within the language server.

Implementation and Finalization In the implementation phase the actual refactoring features are implemented according to the analysis. To make the implementation ready for contribution it needs to be refined, which means the code needs to be readable and follow the development guidelines [19]. When the refinement is done, a pull request can be created to contribute the changes upstream and to finalize the implementation step.

7.2. Project Plan

The project plan was developed in week 3 after the ideas for the refactoring features were set. Without concrete ideas a good project plan can not be worked out. The documentation part is spread over the whole project duration as it is important to keep it up to date and consistent. All implementation work should be completed two weeks before the deadline to ensure that the documentation is finished and ready for submission.

Figure 22 shows the project plan (on top, in lighter hue) compared to the actual progress made (on the bottom, in darker hue). There is almost no deviation from the plan, except for the first refactoring, which was implemented faster than expected, and the documentation, where the abstract and management summary were written before the refinement phase, because of the earlier submission date.

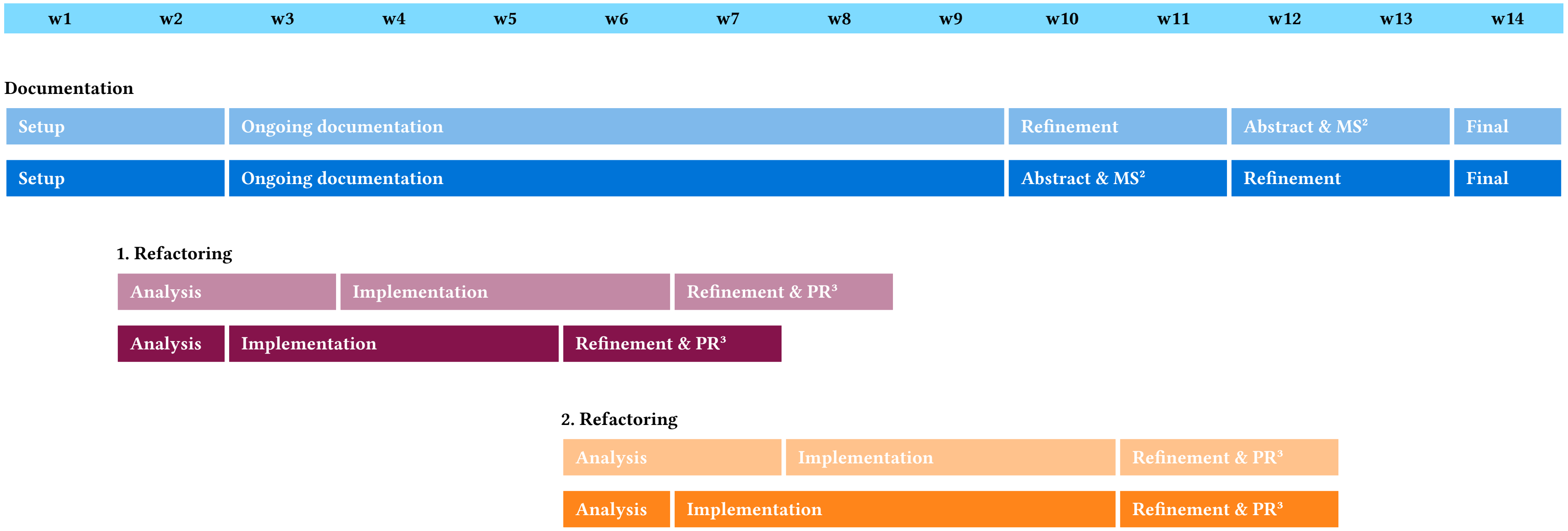


Figure 22: Project plan

[2]Management Summary

[3]Pull Request

7.3. Time Tracking

To monitor the working hours effectively, a Google Sheet was established where information about the time spent was meticulously recorded. Each record contains a date, the amount of time spent, name of executor, task category and a brief comment detailing the specific work performed during that time.

Table 11 and Figure 23 show the total time spent on each category per project week. Figure 24 shows the share of time invested per category. Most of the time spent was invested into the documentation and implementation, with the former being the main focus at the end of the project.

Figure 25 shows the time spent by each project author. Both authors contributed a similar amount to both documentation and implementation.

SUM of Hours	*Type*					
Week	Admin	Documentation	Implementation	Meeting	Studying / Research	Grand Total
Week 39	12.3h	4.8h		2.0h	7.5h	26.5h
Week 40	5.5h	3.0h	11.0h	1.0h	11.5h	32.0h
Week 41	0.8h	5.0h	24.5h	2.5h	1.5h	34.3h
Week 42	0.5h	2.8h	9.5h	2.0h	2.0h	16.8h
Week 43	0.5h	8.5h	23.0h	1.0h		33.0h
Week 44		18.0h	2.0h	4.0h	1.5h	25.5h
Week 45		5.5h	24.0h	4.0h	0.5h	34.0h
Week 46		16.0h	12.0h	1.0h		29.0h
Week 47		18.0h	10.0h	4.0h		32.0h
Week 48		26.0h	3.0h	2.0h		31.0h
Week 49		18.5h	5.0h	1.0h		24.5h
Week 50	1.5h	34.0h	3.0h	1.0h		39.5h
Week 51	0.5h	37.5h		1.0h		39.0h
Week 52	4.0h	45.5h		1.0h		50.5h
Grand Total	**25.5h**	**243.0h**	**127.0h**	**27.5h**	**24.5h**	**447.5h**

Table 11: Time invested per category and project week

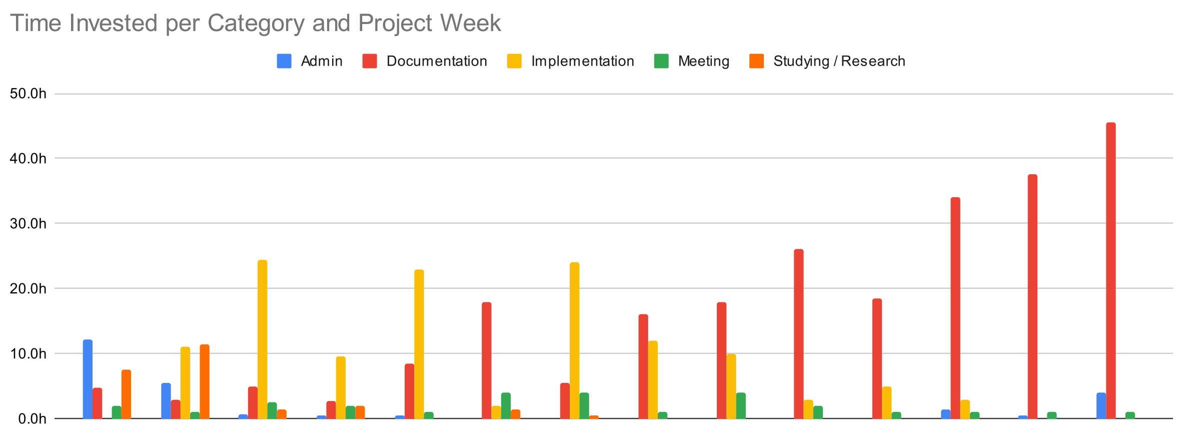


Figure 23: Time invested per category and project week

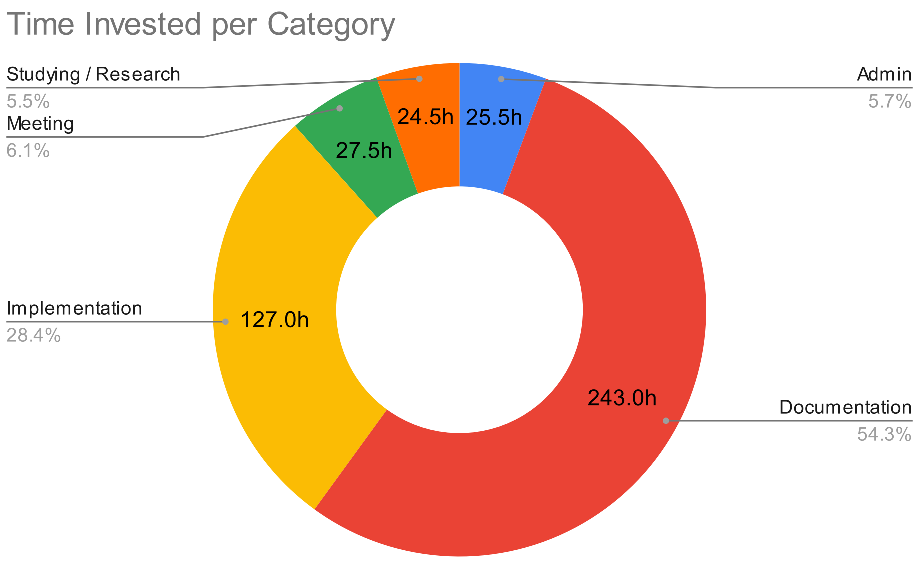


Figure 24: Time invested per category

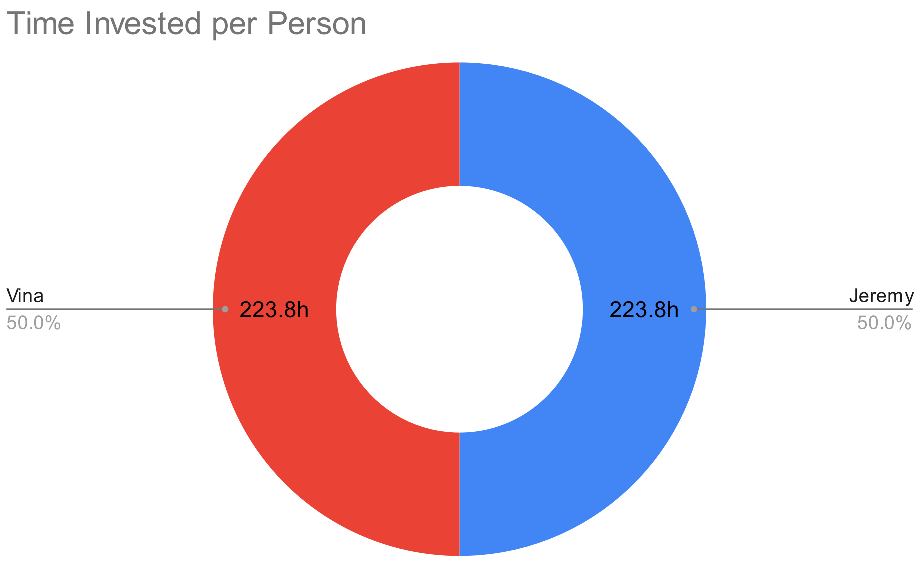


Figure 25: Time invested per person

8. Conclusion

To summarize this thesis this conclusion describes the most important parts of the project. *Section 8.1, Learnings* looks into what was learned working on this project. In *Section 8.2, Outlook* it is described how this project could be extended and what the future of the language server looks like.

In this thesis, it was analyzed how the clangd language server works and how additional refactoring features can be added.

When first looking at the LLVM project it can be overwhelming at first but once it is clear how it is put together it is easy to get around with.

Two new refactoring features were implemented according to the analysis and were submitted to LLVM (pull requests are opened). The intention for these implementation is to be contributed to the clangd language server which should help C++ 20 developers to work with the concepts for which there was no refactoring beforehand.

Inline Concept Requirement Transforms the concept containing a `requires` clause into a restricted function template. This transformation results in a less complex and shorter code.

Abbreviate Function Template Transforms the function template to its abbreviated form. The template parameter types are replaced with the `auto` keyword where possible. This transformation results in less code as the `template` declaration will be removed.

Unfortunately, the opened pull requests were not merged until this thesis was handed in therefore the added refactorings are not yet available in the current version of clangd.

8.1. Learnings

Overall this project was very interesting and enriched working on an open source project like LLVM.

For future projects, it should be considered to spend more time on the analysis part. The implementation was started before all the necessary research was done, resulting in some avoidable pitfalls and slowing down the development process. Most of these cases came up during the second refactoring *Section 5, Refactoring — Abbreviate Function Template* and were related to parameter types that were not considered, such as array types and function pointers. Those issues could probably have been avoided if more time were spent analyzing which parameter types exist and properly documenting them.

Also, the documentation part of this project was underestimated a lot and it would have been good to start documenting the analysis part in the beginning. Finding a good structure for the documentation was hard and it was changed many times.

8.2. Outlook

The content of this project could have been extended a lot more as there are almost unlimited options to add new refactorings. As long as the C++ language is evolving more refactorings can be added to help support the developers. In the case of this thesis, an option would have been possible to switch between different forms of concepts. That feature could have connected the two refactoring implemented but this also would have taken way longer than it already did.

The LLVM project is an active open source project that receives a lot of pull requests each day. This is giving hope, that it will continue to grow in the future.

Hopefully, the open pull requests will be accepted so the new refactorings will be available on the clangd language server. But it remains unclear if the usage of language servers will increase or if IDEs are going back to implementing their own code support as JetBrains has already announced that they might. [55]

9. Disclaimer

Parts of this paper were rephrased using GPT-3.5 [56] and GPT-4 [57]. This paper was reviewed by the advisor and corrections were applied according to the comments made. For spell checking and translations DeepL [58], LanguageTool [59], grammarly [60] and QuillBot [61] was used.

10. Glossary

AST Abbreviation for Abstract Syntax Tree. It is a hierarchical representation of source code structure, capturing syntax and semantics.

IDE Integrated Development Environment. Software application that provies comprehensive facilities for software development.

LLVM Abbreviation for Low Level Virtual Machine. A set of compiler and toolchain technologies.

Tweak In the context of the clangd language server, it is a small, context-sensitive refactoring.

11. Bibliography

[1] J. Stucki and V. Zahnd, *Title Image (AI Generated, Bing, Prompt: "Old newspaper announcing C++ concepts").*

[2] “ISO/IEC TS 19217:2015”. Accessed: Dec. 15, 2023. [Online]. Available: https://www.iso.org/standard/64031.html

[3] D. Exterman, “Tips for C++ Refactoring”. Accessed: Dec. 14, 2023. [Online]. Available: https://www.incredibuild.com/blog/tips-for-c-refactoring

[4] C. Foundation, “The Committee”. Accessed: Nov. 17, 2023. [Online]. Available: https://isocpp.org/std/the-committee

[5] “Constraints and concepts”. Accessed: Nov. 16, 2023. [Online]. Available: https://en.cppreference.com/w/cpp/language/constraints

[6] A. S. Gillis, “refactoring”. Accessed: Dec. 16, 2023. [Online]. Available: https://www.techtarget.com/searchapparchitecture/definition/refactoring

[7] B. Ltd, “Language Server Protocol (LSP)”. Accessed: Dec. 14, 2023. [Online]. Available: https://www.bitservices.io/blog/language-server-protocol/

[8] “The LLVM Project”. Accessed: Oct. 23, 2023. [Online]. Available: https://github.com/llvm/llvm-project

[9] M. Fowler, “Refactoring”. Accessed: Dec. 18, 2023. [Online]. Available: https://refactoring.com/

[10] “Preprocessor”. Accessed: Dec. 20, 2023. [Online]. Available: https://en.cppreference.com/w/cpp/preprocessor

[11] Microsoft, “Language Server Protocol”. Accessed: Nov. 17, 2023. [Online]. Available: https://microsoft.github.io/language-server-protocol

[12] Microsoft, “What is the Language Server Protocol?”. Accessed: Nov. 17, 2023. [Online]. Available: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview

[13] Microsoft, “Tools supporting the LSP”. Accessed: Nov. 24, 2023. [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/tools/

[14] Microsoft, “Implementations”. Accessed: Nov. 17, 2023. [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/servers/

[15] “WorkspaceEdit”. Accessed: Nov. 10, 2023. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#workspaceEdit

[16] “Applies a WorkspaceEdit”. Accessed: Nov. 10, 2023. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#workspace_applyEdit

[17] J. Mangiavacchi, “Swift, C, LLVM Compiler Optimization”. Accessed: Oct. 10, 2023. [Online]. Available: https://medium.com/@JMangia/swift-c-llvm-compiler-optimization-842012568bb7

[18] “What is clangd?”. Accessed: Nov. 17, 2023. [Online]. Available: https://clangd.llvm.org/

[19] “LLVM Coding standards”. Accessed: Nov. 10, 2023. [Online]. Available: https://llvm.org/docs/CodingStandards.html

[20] “Function Declaration”. Accessed: Nov. 20, 2023. [Online]. Available: https://en.cppreference.com/w/cpp/language/function

[21] T. C. Team, “Clang-Tidy”. [Online]. Available: https://clang.llvm.org/extra/clang-tidy/

[22] T. C. Team, “clangFormat”. [Online]. Available: https://clang.llvm.org/docs/ClangFormat.html

[23] T. C. Team, “Clangd Testing”. Accessed: Nov. 10, 2023. [Online]. Available: https://clangd.llvm.org/design/code.html#testing

[24] “googletest Framework”. Accessed: Nov. 10, 2023. [Online]. Available: https://github.com/google/googletest

[25] “clang::clangd::Tweak Class Reference”. [Online]. Available: https://clang.llvm.org/extra/doxygen/classclang_1_1clangd_1_1Tweak.html

[26] B. P. C, “Abstract Syntax Tree (AST) - Explained in Plain English ”. Accessed: Dec. 20, 2023. [Online]. Available: https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38

[27] A. Finamore, “Abstract Syntax Trees in Python”. Accessed: Nov. 20, 2023. [Online]. Available: https://pybit.es/articles/ast-intro/

[28] “Introduction to the Clang AST”. Accessed: Nov. 25, 2023. [Online]. Available: https://clang.llvm.org/docs/IntroductionToTheClangAST.html

[29] “clang”. Accessed: Dec. 20, 2023. [Online]. Available: https://clang.llvm.org/doxygen

[30] “clang::TranslationUnitDecl Class Reference”. Accessed: Nov. 26, 2023. [Online]. Available: https://clang.llvm.org/doxygen/classclang_1_1TranslationUnitDecl.html

[31] “clang::FunctionDecl Class Reference”. Accessed: Nov. 26, 2023. [Online]. Available: https://clang.llvm.org/doxygen/classclang_1_1FunctionDecl.html

[32] “C++ keyword: requires”. Accessed: Nov. 27, 2023. [Online]. Available: https://en.cppreference.com/w/cpp/keyword/requires

[33] “C++ keyword: concept”. Accessed: Nov. 27, 2023. [Online]. Available: https://en.cppreference.com/w/cpp/keyword/concept

[34] B. Filipek, “C++20 Concepts - a Quick Introductio ”. Accessed: Nov. 26, 2023. [Online]. Available: https://www.cppstories.com/2021/concepts-intro/

[35] “std::enable_if”. Accessed: Dec. 21, 2023. [Online]. Available: https://en.cppreference.com/w/cpp/types/enable_if

[36] B. Filipek, “Simplify Code with if constexpr and Concepts in C++17/C++20 ”. Accessed: Nov. 26, 2023. [Online]. Available: https://www.cppstories.com/2018/03/ifconstexpr

[37] “Allow CodeActions to specify cursor position”. Accessed: Nov. 09, 2023. [Online]. Available: https://github.com/microsoft/language-server-protocol/issues/724

[38] “How can a server implement "extract method" (how can it trigger rename?)”. Accessed: Nov. 09, 2023. [Online]. Available: https://github.com/microsoft/language-server-protocol/issues/764

[39] J. Stucki and V. Zahnd, “[clangd] Add tweak to inline concept requirements”. Accessed: Oct. 23, 2023. [Online]. Available: https://github.com/llvm/llvm-project/pull/69693

[40] J. Stucki and V. Zahnd, “[clangd] Add tweak to abbreviate function templates”. Accessed: Dec. 07, 2023. [Online]. Available: https://github.com/llvm/llvm-project/pull/74710

[41] “The LLVM Project”. Accessed: Dec. 16, 2023. [Online]. Available: https://github.com/sa-concept-refactoring/llvm-projectt

[42] Accessed: Dec. 18, 2023. [Online]. Available: https://clangd.llvm.org/logo.svg

[43] “Brand Assets”. Accessed: Dec. 18, 2023. [Online]. Available: https://www.jetbrains.com/company/brand/

[44] “Git Logos”. Accessed: Dec. 18, 2023. [Online]. Available: https://git-scm.com/downloads/logos

[45] “GitHub Logos and Usage”. Accessed: Dec. 18, 2023. [Online]. Available: https://github.com/logos

[46] “LLVM Logo”. Accessed: Dec. 18, 2023. [Online]. Available: https://www.llvm.org/Logo.html

[47] “Icons and names usage guidelines”. Accessed: Dec. 18, 2023. [Online]. Available: https://code.visualstudio.com/brand

[48] llvm, “clangd extension”. Accessed: Oct. 23, 2023. [Online]. Available: https://github.com/clangd/vscode-clangd

[49] Microsoft, “CMake extension”. Accessed: Oct. 26, 2023. [Online]. Available: https://github.com/microsoft/vscode-cmake-tools

[50] “VSCodium”. Accessed: Oct. 23, 2023. [Online]. Available: https://github.com/VSCodium/vscodium

[51] “Neovim”. Accessed: Dec. 12, 2023. [Online]. Available: https://neovim.io/

[52] “Typst”. Accessed: Dec. 12, 2023. [Online]. Available: https://typst.app/

[53] “Typst LSP”. Accessed: Dec. 20, 2023. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=nvarner.typst-lsp

[54] “Project planning for developers”. Accessed: Dec. 20, 2023. [Online]. Available: https://github.com/features/issues

[55] A. Kazakova, “CLion Nova Explodes onto the C and C++ Development Scene”. Accessed: Dec. 03, 2023. [Online]. Available: https://blog.jetbrains.com/clion/2023/11/clion-nova

[56] “ChatGPT 3.5”. Accessed: Dec. 21, 2023. [Online]. Available: https://chat.openai.com/

[57] “ChatGPT 4”. Accessed: Dec. 21, 2023. [Online]. Available: https://chat.openai.com/

[58] “DeepL”. Accessed: Dec. 21, 2023. [Online]. Available: https://www.deepl.com/translator

[59] “LanguageTool”. Accessed: Dec. 21, 2023. [Online]. Available: https://languagetool.org/

[60] “Grammarly”. Accessed: Dec. 21, 2023. [Online]. Available: https://app.grammarly.com/

[61] “QuillBot”. Accessed: Dec. 21, 2023. [Online]. Available: https://quillbot.com/grammar-check

12. Table of Figures

Figure 1: Artistic interpretation of this project [1] 1
Figure 2: Diagram showing integration of implemented refactoring iii
Figure 3: Diagram showing example communication between IDE and language server [12] 6
Figure 4: Diagram showing code action and code action resolve request 7
Figure 5: Diagram showing the architecture of LLVM [17] 9
Figure 6: Class digram for clangd tweak and refactor example 10
Figure 7: Diagram showing tree structure 12
Figure 8: Diagram showing steps for code generation [27] 13
Figure 9: Functions using requires clause 15
Figure 10: Elements captured for the "Inline Concept Requirement" refactoring 21
Figure 11: Example AST tranformation of the "Inline Concept Requirement" refactoring 22
Figure 12: Example for "Inline Concept Requirement" refactoring with multiple type arguments 25
Figure 13: Screenshot showing the option to inline a concept requirement in VS Code 26
Figure 14: Screenshot showing the option to inline a concept requirement in Neovim 26
Figure 15: Elements captured for the "Abbreviate Function Template" refactoring 29
Figure 16: Example to illustrate call site differences of auto parameters 29
Figure 17: Example AST tranformation of "Abbreviate Function Template" refactoring 30
Figure 18: Screenshot showing the option to abbreviate a function template in VS Code 33
Figure 19: Screenshot showing the option to abbreviate a function template in Neovim 33
Figure 20: Diagram showing structure and workflow of the project
(Logos from [42], [43], [44], [45], [46], [47]) 34
Figure 21: Screenshot showing build settings in CLion on Windows 36
Figure 22: Project plan 38
Figure 23: Time invested per category and project week 39
Figure 24: Time invested per category 40
Figure 25: Time invested per person 40

13. Table of Tables

Table 1: Example of “Inline Concept Requirement” refactoring ii
Table 2: Example of “Abbreviate Function Template” refactoring ii
Table 3: A bad example of refactoring 5
Table 4: C++ language servers implementing the LSP [14] 7
Table 5: AST dump possibilities for valid and invalid functions 14
Table 6: Different ways to constrain a function template using concepts 16
Table 7: Capabilities of the “Inline Concept Requirement” refactoring 19
Table 8: Checks made during the preparation phase of the "Inline Concept Requirement" refactoring 20
Table 9: Capabilities of the “Abbreviate Function Template” refactoring 27
Table 10: Checks made during the preparation phase of the "Abbreviate Function Template" refactoring 28
Table 11: Time invested per category and project week 39

14. List of Listings

Listing 1: Example answer to a code action request 8
Listing 2: Structure of a tweak class 11
Listing 3: Function for calculating the factorial of a number 12
Listing 4: Example of AST dump in clang [28] 13
Listing 5: Requires clause using disjunction 15
Listing 6: Requires clause using conjunction 15
Listing 7: Hashable concept declaration 15
Listing 8: An existing conjunction 17
Listing 9: The proposed refactoring to the conjunction in Listing 8 17
Listing 10: Simple function template declaration 18
Listing 11: Listing 10 with the concept extracted 18
Listing 12: Combined concept requirement 24
Listing 13: Class template 24
Listing 14: Templated function parameter 32
Listing 15: Template argument used for multiple function parameters 32
Listing 16: Function template with requires clause 32
Listing 17: Configuration in settings.json file for to VS Code 35
Listing 18: Configuring clangd in Neovim 35

15. Appendix

In this last section the personal reports from both authors can be found in which they reflect on the project (*Section 15.1, Personal Report — Jeremy Stucki* and *Section 15.2, Personal Report — Vina Zahnd*). It also contains the final version of the code written during this project (*Section 15.4, Source Code*), which includes the implemented refactorings and the test project.

The assignment given can be found in *Section 15.3, Assignment*.

15.1. Personal Report — Jeremy Stucki

Contributing to an open source project as part of this SA was very exciting. I am really looking forward to our features getting merged and released as part of clangd. It was really helpful that I already have experience working with GitHub and open source projects.

Actually programming was quite tedious at times due to builds taking a long time and useful functions not being easily discoverable. Also, some advanced language features could not be used, as clangd itself is using C++17. This means that, funnily enough, concepts could not be used. However, in the end, the actual code required to add the refactorings was quite simple. There was no need to adjust large configuration files, and the abstraction from the language server protocol felt at the right level.

Like with most projects my biggest struggle was the documentation. What helped a lot was restructuring the outline early on. I also had a pretty hard time finding the appropriate tone to write in. My highlight was that I got to try out a new documentation tool (Typst [52]), which made writing the documentation quite a bit more fun than LaTeX.

As for teamwork, Vina and I worked great together, and I had a lot of fun. Especially towards the end, when we had a great workflow with pull requests and issues to work on the project asynchronously. The weekly meetings with our advisor were also really helpful, since he has a very deep understanding of C++.

In conclusion, I found the project to be a good learning experience, and I now know a lot more about the language server protocol. It feels like the time was spent well, because we were able to submit the entire work upstream, where it will hopefully live on.

15.2. Personal Report — Vina Zahnd

Contributing to an open source project like LLVM was a great experience even though the changes are not yet merged. For me this was the first time to work on an open source project and I learned a lot while doing so.

At first it seemed to be a very difficult project to work with, but we were able to get it running quickly. I was struggling a lot with building the project in the beginning as it took a lot of memory to do so and would end in errors. After a few trials I decided to not close my notebook anymore until it has built completely, which worked out in the end. Writing the code itself required a lot of trial and error as well as reading the clangd documentation. When the refactoring feature was finally displaying in the VS Code editor it felt like a huge achievement.

The documentation part of this project was hard to get into and included a lot of procrastination. We were struggling a lot with the structure of the documentation and what content we should include.

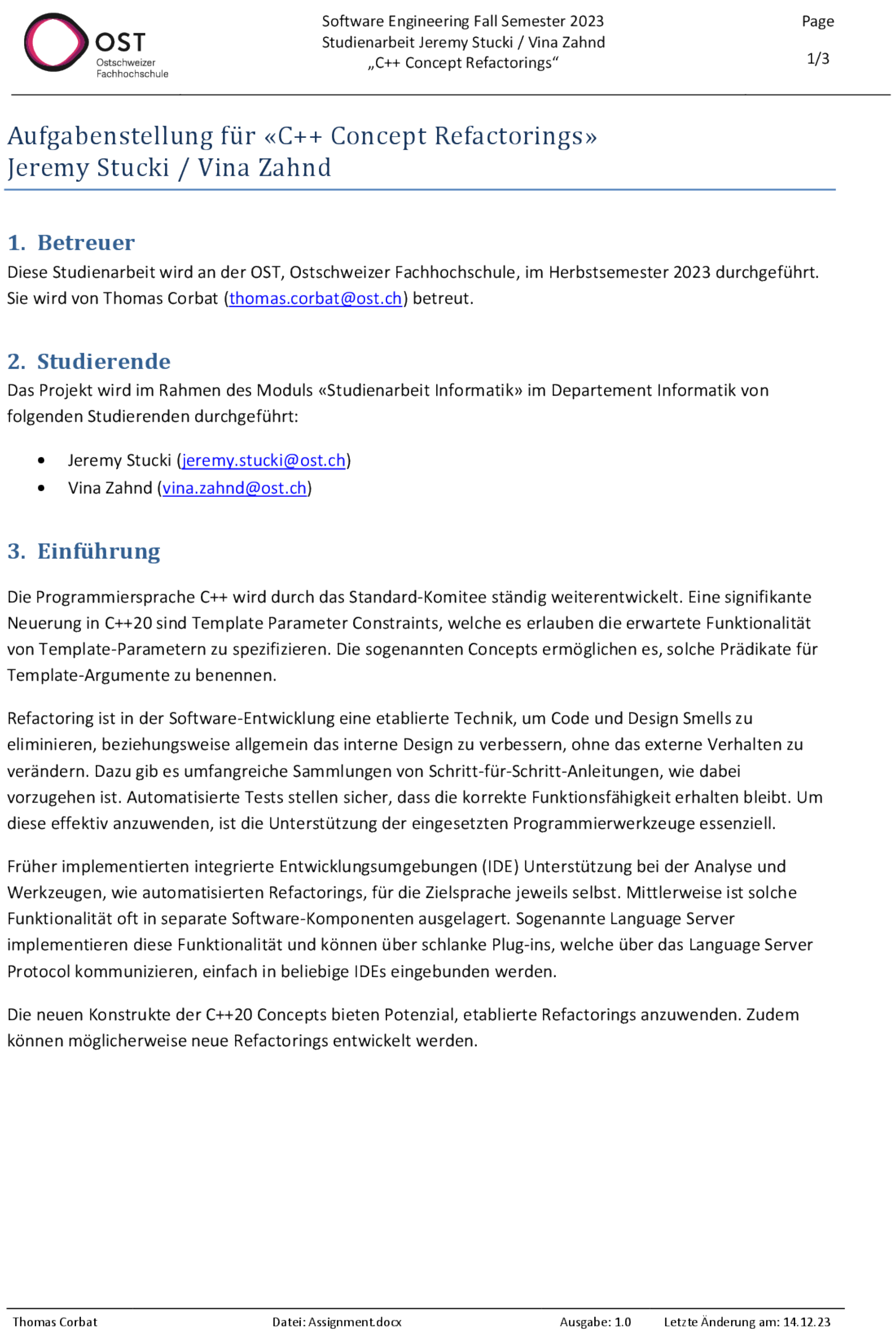
Jeremy and I were a good team and got along really good. Whenever we had a disagreement, we were able to solve it in a good manner. As Jeremy is more experienced in C++ he needed to help me from time to time and was way faster then me in making development progress. He explained certain things which was a good addition to my knowledge and was appreciated.

Our advisor was always friendly and helpful at all times. We had good discussions within our weekly meetings and I felt to be in good hands.

My personal highlight was when we first got comments on our Pull Request. The reviewer left us some very good comments and wrote that he is looking forward to having the new refactoring feature within the project.

To conclude, it was a good project overall and a good addition to my career and I am even motivated to contribute further to the clangd language server or any other open source project.

15.3. Assignment



OST
Ostschweizer Fachhochschule

Software Engineering Fall Semester 2023
Studienarbeit Jeremy Stucki / Vina Zahnd
„C++ Concept Refactorings“

Page 1/3

Aufgabenstellung für «C++ Concept Refactorings» Jeremy Stucki / Vina Zahnd

1. Betreuer

Diese Studienarbeit wird an der OST, Ostschweizer Fachhochschule, im Herbstsemester 2023 durchgeführt. Sie wird von Thomas Corbat (thomas.corbat@ost.ch) betreut.

2. Studierende

Das Projekt wird im Rahmen des Moduls «Studienarbeit Informatik» im Departement Informatik von folgenden Studierenden durchgeführt:

- Jeremy Stucki (jeremy.stucki@ost.ch)
- Vina Zahnd (vina.zahnd@ost.ch)

3. Einführung

Die Programmiersprache C++ wird durch das Standard-Komitee ständig weiterentwickelt. Eine signifikante Neuerung in C++20 sind Template Parameter Constraints, welche es erlauben die erwartete Funktionalität von Template-Parametern zu spezifizieren. Die sogenannten Concepts ermöglichen es, solche Prädikate für Template-Argumente zu benennen.

Refactoring ist in der Software-Entwicklung eine etablierte Technik, um Code und Design Smells zu eliminieren, beziehungsweise allgemein das interne Design zu verbessern, ohne das externe Verhalten zu verändern. Dazu gib es umfangreiche Sammlungen von Schritt-für-Schritt-Anleitungen, wie dabei vorzugehen ist. Automatisierte Tests stellen sicher, dass die korrekte Funktionsfähigkeit erhalten bleibt. Um diese effektiv anzuwenden, ist die Unterstützung der eingesetzten Programmierwerkzeuge essenziell.

Früher implementierten integrierte Entwicklungsumgebungen (IDE) Unterstützung bei der Analyse und Werkzeugen, wie automatisierten Refactorings, für die Zielsprache jeweils selbst. Mittlerweise ist solche Funktionalität oft in separate Software-Komponenten ausgelagert. Sogenannte Language Server implementieren diese Funktionalität und können über schlanke Plug-ins, welche über das Language Server Protocol kommunizieren, einfach in beliebige IDEs eingebunden werden.

Die neuen Konstrukte der C++20 Concepts bieten Potenzial, etablierte Refactorings anzuwenden. Zudem können möglicherweise neue Refactorings entwickelt werden.

Thomas Corbat | Datei: Assignment.docx | Ausgabe: 1.0 | Letzte Änderung am: 14.12.23

4. Ziele des Projektes

In dieser Studienarbeit sollen mehrere Aspekte von möglichen C++20-Concept Refactorings durch die Studierenden bearbeitet werden:

- Allgemein bekannte Refactorings können möglicherweise auf Concepts angewendet werden. Dies ist möglicherweise bereits in aktuell verfügbarem Tooling implementiert.
- Die Concepts bieten Potenzial für neue Refactorings. Falls solche identifiziert werden können, sind diese zu definieren. Hier besteht Potenzial für ein wissenschaftliches Paper.
- Der clangd Language Server soll um automatisierte Code Transformationsfunktionalität erweitert werden, um als Proof-of-Concept der vorgängig identifizierten Concept-Refactorings zu dienen.

Da die ersten beiden Punkte explorativen Charakter haben, können die genauen Ziele der Arbeit vorgängig nicht exakt definiert werden. Es ist keine Pflichtanforderung, dass die implementierten Features für clangd auch in das Produkt übernommen werden, da dies teilweise ausserhalb des Einflussbereichs der Studierenden liegt.

5. Dokumentation

Das Projekt muss entsprechend der Richtlinien des Departments Informatik dokumentiert werden [1]. Dies umfasst alle Analyse, Design, Implementation und Projekt Management, usw. Kapitel. Die Dokumentation sollte vorzugsweise auf Englisch verfasst werden. Ein Projektplan muss am Anfang ausgearbeitet und entsprechend dem effektiven Fortschritt angepasst werden. Schlussendlich müssen alle Resultate komplett auf den Archiv-Server der OST hochgeladen werden.

Eine gedruckte Ausgabe der Dokumentation muss dem Betreuer abgegeben werden (farbig und doppelseitig gedruckt, gebunden).

6. Wichtige Daten

* Die URLs könnten noch angepasst werden.

18.09.2023	**Semesterstart und Beginn des Projektes**
Bis 21.12.2023	**Erfassen des Abstracts im Online-Tool und Überprüfung durch den Betreuer [2]**
22.12.2023, 17:00 Uhr	**Finale Abgabe über den Archiv-Server [3]**

7. Bewertung

Bei erfolgreichem Bestehen der Studienarbeit erhalten die Studierenden 8 ECTS-Punkte. Der geschätzte Aufwand für ein ECTS beträgt 30 Arbeitsstunden (vgl. Modulbeschreibung [4]). Der Betreuer ist verantwortlich für die Bewertung der geleisteten Arbeit.

Kriterium	Gewicht
1. Organisation, Ausführung	1/5
2. Bericht (Abstract, Management Summary, technischer und persönlicher Bericht) sowie Struktur, Visualisierung und Sprache der ganzen Dokumentation.	1/5
3. Inhalt der Arbeit	3/5

Weiter gelten die generellen Regularien für Studienarbeiten im Departement Informatik.

8. Referenzen

[1] https://ostch.sharepoint.com/:b:/r/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten/Informationen/Leitfaden%20BA%20SA%20v1.0.pdf?csf=1&web=1&e=1qAL6o

[2] https://abstract.rj.ost.ch

[3] https://avt.i.ost.ch

[4] https://studien.rj.ost.ch/allModules/24386_M_SAI14.html

Rapperswil, 17. September 2023

Thomas Corbat

Lehrbeauftragter
OST – Ostschweizer Fachhochschule

15.4. Source Code

Inline Concept Requirement Refactoring — InlineConceptRequirement.cpp 58
Inline Concept Requirement Refactoring — InlineConceptRequirementTests.cpp 60
Abbreviate Function Template Refactoring — AbbreviateFunctionTemplate.cpp 61
Abbreviate Function Template Refactoring — AbbreviateFunctionTemplateTests.cpp 65
Test Project — InlineConceptRequirement.cxx 66
Test Project — AbbreviateFunctionTemplate.cxx 67

Inline Concept Requirement Refactoring — InlineConceptRequirement.cpp

```cpp
//===--- InlineConceptRequirement.cpp -----------------------------*- C++-*-===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//
#include "ParsedAST.h"
#include "SourceCode.h"
#include "refactor/Tweak.h"
#include "support/Logger.h"
#include "clang/AST/ASTContext.h"
#include "clang/AST/ExprConcepts.h"
#include "clang/Tooling/Core/Replacement.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/Support/Casting.h"
#include "llvm/Support/Error.h"

namespace clang {
namespace clangd {
namespace {
/// Inlines a concept requirement.
///
/// Before:
///   template <typename T> void f(T) requires foo<T> {}
///                                            ^^^^^^
/// After:
///   template <foo T> void f(T) {}
class InlineConceptRequirement : public Tweak {
public:
  const char *id() const final;

  auto prepare(const Selection &Inputs) -> bool override;
  auto apply(const Selection &Inputs) -> Expected<Effect> override;
  auto title() const -> std::string override {
    return "Inline concept requirement";
  }
  auto kind() const -> llvm::StringLiteral override {
    return CodeAction::REFACTOR_KIND;
  }

private:
  const ConceptSpecializationExpr *ConceptSpecializationExpression;
  const TemplateTypeParmDecl *TemplateTypeParameterDeclaration;
  const syntax::Token *RequiresToken;

  static auto getTemplateParameterIndexOfTemplateArgument(
      const TemplateArgument &TemplateArgument) -> std::optional<int>;
  auto generateRequiresReplacement(ASTContext &)
      -> llvm::Expected<tooling::Replacement>;
  auto generateRequiresTokenReplacement(const syntax::TokenBuffer &)
      -> tooling::Replacement;
  auto generateTemplateParameterReplacement(ASTContext &Context)
      -> llvm::Expected<tooling::Replacement>;

  static auto findToken(const ParsedAST *, const SourceRange &,
                        const tok::TokenKind) -> const syntax::Token *;

  template <typename T, typename NodeKind>
  static auto findNode(const SelectionTree::Node &Root)
      -> std::tuple<const T *, const SelectionTree::Node *>;

  template <typename T>
  static auto findExpression(const SelectionTree::Node &Root)
      -> std::tuple<const T *, const SelectionTree::Node *> {
    return findNode<T, Expr>(Root);
  }

  template <typename T>
  static auto findDeclaration(const SelectionTree::Node &Root)
      -> std::tuple<const T *, const SelectionTree::Node *> {
    return findNode<T, Decl>(Root);
  }
};

REGISTER_TWEAK(InlineConceptRequirement)

auto InlineConceptRequirement::prepare(const Selection &Inputs) -> bool {
  // Check if C++ version is 20 or higher
  if (!Inputs.AST->getLangOpts().CPlusPlus20)
    return false;

  const auto *Root = Inputs.ASTSelection.commonAncestor();
  if (!Root)
    return false;

  const SelectionTree::Node *ConceptSpecializationExpressionTreeNode;
  std::tie(ConceptSpecializationExpression,
           ConceptSpecializationExpressionTreeNode) =
      findExpression<ConceptSpecializationExpr>(*Root);
  if (!ConceptSpecializationExpression)
```

```
    return false;

  // Only allow concepts that are direct children of function template
  // declarations or function declarations. This excludes conjunctions of
  // concepts which are not handled.
  const auto *ParentDeclaration =
      ConceptSpecializationExpressionTreeNode->Parent->ASTNode.get<Decl>();
  if (!isa_and_nonnull<FunctionTemplateDecl>(ParentDeclaration) &&
      !isa_and_nonnull<FunctionDecl>(ParentDeclaration))
    return false;

  const FunctionTemplateDecl *FunctionTemplateDeclaration =
      std::get<0>(findDeclaration<FunctionTemplateDecl>(*Root));
  if (!FunctionTemplateDeclaration)
    return false;

  auto TemplateArguments =
      ConceptSpecializationExpression->getTemplateArguments();
  if (TemplateArguments.size() != 1)
    return false;

  auto TemplateParameterIndex =
      getTemplateParameterIndexOfTemplateArgument(TemplateArguments[0]);
  if (!TemplateParameterIndex)
    return false;

  TemplateTypeParameterDeclaration = dyn_cast_or_null<TemplateTypeParmDecl>(
      FunctionTemplateDeclaration->getTemplateParameters()->getParam(
          *TemplateParameterIndex));
  if (!TemplateTypeParameterDeclaration->wasDeclaredWithTypename())
    return false;

  RequiresToken =
      findToken(Inputs.AST, FunctionTemplateDeclaration->getSourceRange(),
                tok::kw_requires);
  if (!RequiresToken)
    return false;

  return true;
}

auto InlineConceptRequirement::apply(const Selection &Inputs)
    -> Expected<Tweak::Effect> {
  auto &Context = Inputs.AST->getASTContext();
  auto &TokenBuffer = Inputs.AST->getTokens();

  tooling::Replacements Replacements{};

  auto TemplateParameterReplacement =
      generateTemplateParameterReplacement(Context);

  if (auto Err = TemplateParameterReplacement.takeError())
    return Err;

  if (auto Err = Replacements.add(*TemplateParameterReplacement))
    return Err;

  auto RequiresReplacement = generateRequiresReplacement(Context);

  if (auto Err = RequiresReplacement.takeError())
    return Err;

  if (auto Err = Replacements.add(*RequiresReplacement))
    return Err;

  if (auto Err =
          Replacements.add(generateRequiresTokenReplacement(TokenBuffer)))
    return Err;

  return Effect::mainFileEdit(Context.getSourceManager(), Replacements);
}

auto InlineConceptRequirement::getTemplateParameterIndexOfTemplateArgument(
    const TemplateArgument &TemplateArgument) -> std::optional<int> {
  if (TemplateArgument.getKind() != TemplateArgument.Type)
    return {};

  auto TemplateArgumentType = TemplateArgument.getAsType();
  if (!TemplateArgumentType->isTemplateTypeParmType())
    return {};

  const auto *TemplateTypeParameterType =
      TemplateArgumentType->getAs<TemplateTypeParmType>();
  if (!TemplateTypeParameterType)
    return {};

  return TemplateTypeParameterType->getIndex();
}

auto InlineConceptRequirement::generateRequiresReplacement(ASTContext &Context)
    -> llvm::Expected<tooling::Replacement> {
  auto &SourceManager = Context.getSourceManager();

  auto RequiresRange =
      toHalfOpenFileRange(SourceManager, Context.getLangOpts(),
                          ConceptSpecializationExpression->getSourceRange());
  if (!RequiresRange)
    return error("Could not obtain range of the 'requires' branch. Macros?");

  return tooling::Replacement(
      SourceManager, CharSourceRange::getCharRange(*RequiresRange), "");
}

auto InlineConceptRequirement::generateRequiresTokenReplacement(
    const syntax::TokenBuffer &TokenBuffer) -> tooling::Replacement {
  auto &SourceManager = TokenBuffer.sourceManager();

  auto Spelling =
      TokenBuffer.spelledForExpanded(llvm::ArrayRef(*RequiresToken));

  auto DeletionRange =
      syntax::Token::range(SourceManager, Spelling->front(), Spelling->back())
          .toCharRange(SourceManager);

  return tooling::Replacement(SourceManager, DeletionRange, "");
}

auto InlineConceptRequirement::generateTemplateParameterReplacement(
    ASTContext &Context) -> llvm::Expected<tooling::Replacement> {
  auto &SourceManager = Context.getSourceManager();

  auto ConceptName = ConceptSpecializationExpression->getNamedConcept()
                         ->getQualifiedNameAsString();

  auto TemplateParameterName =
      TemplateTypeParameterDeclaration->getQualifiedNameAsString();

  auto TemplateParameterReplacement = ConceptName + ' ' + TemplateParameterName;

  auto TemplateParameterRange =
      toHalfOpenFileRange(SourceManager, Context.getLangOpts(),
                          TemplateTypeParameterDeclaration->getSourceRange());

  if (!TemplateParameterRange)
    return error("Could not obtain range of the template parameter. Macros?");

  return tooling::Replacement(
      SourceManager, CharSourceRange::getCharRange(*TemplateParameterRange),
      TemplateParameterReplacement);
}
```

```
auto clang::clangd::InlineConceptRequirement::findToken(
    const ParsedAST *AST, const SourceRange &SourceRange,
    const tok::TokenKind TokenKind) -> const syntax::Token * {
  auto &TokenBuffer = AST->getTokens();
  const auto &Tokens = TokenBuffer.expandedTokens(SourceRange);

  const auto Predicate = [TokenKind](const auto &Token) {
    return Token.kind() == TokenKind;
  };

  auto It = std::find_if(Tokens.begin(), Tokens.end(), Predicate);

  if (It == Tokens.end())
    return nullptr;

  return It;
}

template <typename T, typename NodeKind>
auto InlineConceptRequirement::findNode(const SelectionTree::Node &Root)
    -> std::tuple<const T *, const SelectionTree::Node *> {

  for (const auto *Node = &Root; Node; Node = Node->Parent) {
    if (const T *Result = dyn_cast_or_null<T>(Node->ASTNode.get<NodeKind>()))
      return {Result, Node};
  }

  return {};
}

} // namespace
} // namespace clangd
} // namespace clang
```

Inline Concept Requirement Refactoring — InlineConceptRequirementTests.cpp

```
//===-- InlineConceptRequirementTests.cpp ----------------------*- C++ -*-===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//
```

```
#include "TweakTesting.h"
#include "gtest/gtest.h"

namespace clang {
namespace clangd {
namespace {

TWEAK_TEST(InlineConceptRequirement);

TEST_F(InlineConceptRequirementTest, Test) {
  Header = R"cpp(
      template <typename T>
      concept foo = true;

      template <typename T>
      concept bar = true;

      template <typename T, typename U>
      concept baz = true;
    )cpp";

  ExtraArgs = {"-std=c++20"};

  //
  // Extra spaces are expected and will be stripped by the formatter.
  //

  EXPECT_EQ(
      apply("template <typename T, typename U> void f(T) requires f^oo<U> {}"),
      "template <typename T, foo U> void f(T)   {}");

  EXPECT_EQ(
      apply("template <typename T, typename U> requires foo<^T> void f(T) {}"),
      "template <foo T, typename U>   void f(T) {}");

  EXPECT_EQ(apply("template <template <typename> class FooBar, typename T> "
                  "void f() requires foo<^T> {}"),
            "template <template <typename> class FooBar, foo T> void f()   {}");

  EXPECT_AVAILABLE(R"cpp(
      template <typename T> void f(T)
        requires ^f^o^o^<^T^> {}
    )cpp");

  EXPECT_AVAILABLE(R"cpp(
      template <typename T> requires ^f^o^o^<^T^>
      void f(T) {}
    )cpp");

  EXPECT_AVAILABLE(R"cpp(
      template <typename T, typename U> void f(T)
        requires ^f^o^o^<^T^> {}
    )cpp");

  EXPECT_AVAILABLE(R"cpp(
      template <template <typename> class FooBar, typename T>
      void foobar() requires ^f^o^o^<^T^>
      {}
    )cpp");

  EXPECT_UNAVAILABLE(R"cpp(
      template <bar T> void f(T)
        requires ^f^o^o^<^T^> {}
    )cpp");

  EXPECT_UNAVAILABLE(R"cpp(
      template <typename T, typename U> void f(T, U)
        requires ^b^a^z^<^T^,^ ^U^> {}
    )cpp");

  EXPECT_UNAVAILABLE(R"cpp(
      template <typename T> void f(T)
        requires ^f^o^o^<^T^>^ ^&^&^ ^b^a^r^<^T^> {}
    )cpp");

  EXPECT_UNAVAILABLE(R"cpp(
      template <typename T>
      concept ^f^o^o^b^a^r = requires(^T^ ^x^) {
        {x} -> ^f^o^o^;
      };
    )cpp");
}

} // namespace
} // namespace clangd
} // namespace clang
```

Abbreviate Function Template Refactoring — AbbreviateFunctionTemplate.cpp

```
//===-- AbbreviateFunctionTemplate.cpp ---------------------------*- C++-*-===//
//
```

```
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//
#include "FindTarget.h"
#include "SourceCode.h"
#include "XRefs.h"
#include "refactor/Tweak.h"
#include "support/Logger.h"
#include "clang/AST/ASTContext.h"
#include "clang/AST/ExprConcepts.h"
#include "clang/Tooling/Core/Replacement.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/Support/Casting.h"
#include "llvm/Support/Error.h"
#include <numeric>

namespace clang {
namespace clangd {
namespace {
/// Converts a function template to its abbreviated form using auto parameters.
/// Before:
///     template <std::integral T>
///     auto foo(T param) { }
///          ^^^^^^^^^^^
/// After:
///     auto foo(std::integral auto param) { }
class AbbreviateFunctionTemplate : public Tweak {
public:
  const char *id() const final;

  auto prepare(const Selection &Inputs) -> bool override;
  auto apply(const Selection &Inputs) -> Expected<Effect> override;

  auto title() const -> std::string override {
    return llvm::formatv("Abbreviate function template");
  }

  auto kind() const -> llvm::StringLiteral override {
    return CodeAction::REFACTOR_KIND;
  }

private:
  static const char *AutoKeywordSpelling;
  const FunctionTemplateDecl *FunctionTemplateDeclaration;
```

```
  struct TemplateParameterInfo {
    const TypeConstraint *Constraint;
    unsigned int FunctionParameterIndex;
    std::vector<tok::TokenKind> FunctionParameterQualifiers;
    std::vector<tok::TokenKind> FunctionParameterTypeQualifiers;
  };

  std::vector<TemplateParameterInfo> TemplateParameterInfoList;

  auto traverseFunctionParameters(size_t NumberOfTemplateParameters) -> bool;

  auto generateFunctionParameterReplacements(const ASTContext &Context)
      -> llvm::Expected<tooling::Replacements>;

  auto generateFunctionParameterReplacement(
      const TemplateParameterInfo &TemplateParameterInfo,
      const ASTContext &Context) -> llvm::Expected<tooling::Replacement>;

  auto generateTemplateDeclarationReplacement(const ASTContext &Context)
      -> llvm::Expected<tooling::Replacement>;

  static auto deconstructType(QualType Type)
      -> std::tuple<QualType, std::vector<tok::TokenKind>,
                    std::vector<tok::TokenKind>>;
};

REGISTER_TWEAK(AbbreviateFunctionTemplate)

const char *AbbreviateFunctionTemplate::AutoKeywordSpelling =
    getKeywordSpelling(tok::kw_auto);

template <typename T>
auto findDeclaration(const SelectionTree::Node &Root) -> const T * {
  for (const auto *Node = &Root; Node; Node = Node->Parent) {
    if (const T *Result = dyn_cast_or_null<T>(Node->ASTNode.get<Decl>()))
      return Result;
  }

  return nullptr;
}

auto getSpellingForQualifier(tok::TokenKind const &Qualifier) -> const char * {
  if (const auto *Spelling = getKeywordSpelling(Qualifier))
    return Spelling;

  if (const auto *Spelling = getPunctuatorSpelling(Qualifier))
    return Spelling;
  return nullptr;
}

bool AbbreviateFunctionTemplate::prepare(const Selection &Inputs) {
  const auto *CommonAncestor = Inputs.ASTSelection.commonAncestor();
  if (!CommonAncestor)
    return false;

  FunctionTemplateDeclaration =
      findDeclaration<FunctionTemplateDecl>(*CommonAncestor);

  if (!FunctionTemplateDeclaration)
    return false;

  auto *TemplateParameters =
      FunctionTemplateDeclaration->getTemplateParameters();

  auto NumberOfTemplateParameters = TemplateParameters->size();
  TemplateParameterInfoList =
      std::vector<TemplateParameterInfo>(NumberOfTemplateParameters);

  // Check how many times each template parameter is referenced.
  // Depending on the number of references it can be checked
  // if the refactoring is possible:
  // - exactly one: The template parameter was declared but never used, which
  //                means we know for sure it doesn't appear as a parameter.
  // - exactly two: The template parameter was used exactly once, either as a
  //                parameter or somewhere else. This is the case we are
  //                interested in.
  // - more than two: The template parameter was either used for multiple
  //                  parameters or somewhere else in the function.
  for (unsigned TemplateParameterIndex = 0;
       TemplateParameterIndex < NumberOfTemplateParameters;
       TemplateParameterIndex++) {
    auto *TemplateParameter =
        TemplateParameters->getParam(TemplateParameterIndex);
    auto *TemplateParameterInfo =
        &TemplateParameterInfoList[TemplateParameterIndex];

    auto *TemplateParameterDeclaration =
        dyn_cast_or_null<TemplateTypeParmDecl>(TemplateParameter);
    if (!TemplateParameterDeclaration)
      return false;

    TemplateParameterInfo->Constraint =
        TemplateParameterDeclaration->getTypeConstraint();
```

```
    auto TemplateParameterPosition = sourceLocToPosition(
        Inputs.AST->getSourceManager(), TemplateParameter->getEndLoc());

    auto FindReferencesLimit = 3;
    auto ReferencesResult =
        findReferences(*Inputs.AST, TemplateParameterPosition,
                       FindReferencesLimit, Inputs.Index);

    if (ReferencesResult.References.size() != 2)
      return false;
  }

  return traverseFunctionParameters(NumberOfTemplateParameters);
}

auto AbbreviateFunctionTemplate::apply(const Selection &Inputs)
    -> Expected<Tweak::Effect> {
  auto &Context = Inputs.AST->getASTContext();
  auto FunctionParameterReplacements =
      generateFunctionParameterReplacements(Context);

  if (auto Err = FunctionParameterReplacements.takeError())
    return Err;

  auto Replacements = *FunctionParameterReplacements;
  auto TemplateDeclarationReplacement =
      generateTemplateDeclarationReplacement(Context);

  if (auto Err = TemplateDeclarationReplacement.takeError())
    return Err;

  if (auto Err = Replacements.add(*TemplateDeclarationReplacement))
    return Err;

  return Effect::mainFileEdit(Context.getSourceManager(), Replacements);
}

auto AbbreviateFunctionTemplate::traverseFunctionParameters(
    size_t NumberOfTemplateParameters) -> bool {
  auto CurrentTemplateParameterBeingChecked = 0u;
  auto FunctionParameters =
      FunctionTemplateDeclaration->getAsFunction()->parameters();

  for (auto ParameterIndex = 0u; ParameterIndex < FunctionParameters.size();
       ParameterIndex++) {
    auto [RawType, ParameterTypeQualifiers, ParameterQualifiers] =
        deconstructType(FunctionParameters[ParameterIndex]->getOriginalType());

    if (!RawType->isTemplateTypeParmType())
      continue;

    auto TemplateParameterIndex =
        dyn_cast<TemplateTypeParmType>(RawType)->getIndex();

    if (TemplateParameterIndex != CurrentTemplateParameterBeingChecked)
      return false;

    auto *TemplateParameterInfo =
        &TemplateParameterInfoList[TemplateParameterIndex];
    TemplateParameterInfo->FunctionParameterIndex = ParameterIndex;
    TemplateParameterInfo->FunctionParameterTypeQualifiers =
        ParameterTypeQualifiers;
    TemplateParameterInfo->FunctionParameterQualifiers = ParameterQualifiers;

    CurrentTemplateParameterBeingChecked++;
  }

  // All defined template parameters need to be used as function parameters
  return CurrentTemplateParameterBeingChecked == NumberOfTemplateParameters;
}

auto AbbreviateFunctionTemplate::generateFunctionParameterReplacements(
    const ASTContext &Context) -> llvm::Expected<tooling::Replacements> {
  tooling::Replacements Replacements;
  for (const auto &TemplateParameterInfo : TemplateParameterInfoList) {
    auto FunctionParameterReplacement =
        generateFunctionParameterReplacement(TemplateParameterInfo, Context);

    if (auto Err = FunctionParameterReplacement.takeError())
      return Err;

    if (auto Err = Replacements.add(*FunctionParameterReplacement))
      return Err;
  }

  return Replacements;
}

auto AbbreviateFunctionTemplate::generateFunctionParameterReplacement(
    const TemplateParameterInfo &TemplateParameterInfo,
    const ASTContext &Context) -> llvm::Expected<tooling::Replacement> {
  auto &SourceManager = Context.getSourceManager();
```

```
  const auto *Function = FunctionTemplateDeclaration->getAsFunction();
  auto *Parameter =
      Function->getParamDecl(TemplateParameterInfo.FunctionParameterIndex);
  auto ParameterName = Parameter->getDeclName().getAsString();

  std::vector<std::string> ParameterTokens{};

  if (const auto *TypeConstraint = TemplateParameterInfo.Constraint) {
    auto *ConceptReference = TypeConstraint->getConceptReference();
    auto *NamedConcept = ConceptReference->getNamedConcept();

    ParameterTokens.push_back(NamedConcept->getQualifiedNameAsString());

    if (const auto *TemplateArgs = TypeConstraint->getTemplateArgsAsWritten()) {
      auto TemplateArgsRange = SourceRange(TemplateArgs->getLAngleLoc(),
                                           TemplateArgs->getRAngleLoc());
      auto TemplateArgsSource = toSourceCode(SourceManager, TemplateArgsRange);
      ParameterTokens.push_back(TemplateArgsSource.str() + '>');
    }
  }

  ParameterTokens.push_back(AutoKeywordSpelling);

  for (const auto &Qualifier :
       TemplateParameterInfo.FunctionParameterTypeQualifiers) {
    ParameterTokens.push_back(getSpellingForQualifier(Qualifier));
  }

  ParameterTokens.push_back(ParameterName);

  for (const auto &Qualifier :
       TemplateParameterInfo.FunctionParameterQualifiers) {
    ParameterTokens.push_back(getSpellingForQualifier(Qualifier));
  }

  auto FunctionTypeReplacementText = std::accumulate(
      ParameterTokens.begin(), ParameterTokens.end(), std::string{},
      [](auto Result, auto Token) { return std::move(Result) + " " + Token; });

  auto FunctionParameterRange = toHalfOpenFileRange(
      SourceManager, Context.getLangOpts(), Parameter->getSourceRange());

  if (!FunctionParameterRange)
    return error("Could not obtain range of the template parameter. Macros?");

  return tooling::Replacement(
      SourceManager, CharSourceRange::getCharRange(*FunctionParameterRange),
      FunctionTypeReplacementText);
}

auto AbbreviateFunctionTemplate::generateTemplateDeclarationReplacement(
    const ASTContext &Context) -> llvm::Expected<tooling::Replacement> {
  auto &SourceManager = Context.getSourceManager();
  auto *TemplateParameters =
      FunctionTemplateDeclaration->getTemplateParameters();

  auto TemplateDeclarationRange =
      toHalfOpenFileRange(SourceManager, Context.getLangOpts(),
                          TemplateParameters->getSourceRange());

  if (!TemplateDeclarationRange)
    return error("Could not obtain range of the template parameter. Macros?");

  auto CharRange = CharSourceRange::getCharRange(*TemplateDeclarationRange);
  return tooling::Replacement(SourceManager, CharRange, "");
}

auto AbbreviateFunctionTemplate::deconstructType(QualType Type)
    -> std::tuple<QualType, std::vector<tok::TokenKind>,
                  std::vector<tok::TokenKind>> {
  std::vector<tok::TokenKind> ParameterTypeQualifiers{};
  std::vector<tok::TokenKind> ParameterQualifiers{};

  if (Type->isIncompleteArrayType()) {
    ParameterQualifiers.push_back(tok::l_square);
    ParameterQualifiers.push_back(tok::r_square);
    Type = Type->castAsArrayTypeUnsafe()->getElementType();
  }

  if (isa<PackExpansionType>(Type))
    ParameterTypeQualifiers.push_back(tok::ellipsis);

  Type = Type.getNonPackExpansionType();

  if (Type->isRValueReferenceType()) {
    ParameterTypeQualifiers.push_back(tok::ampamp);
    Type = Type.getNonReferenceType();
  }

  if (Type->isLValueReferenceType()) {
    ParameterTypeQualifiers.push_back(tok::amp);
    Type = Type.getNonReferenceType();
  }

  if (Type.isConstQualified()) {
    ParameterTypeQualifiers.push_back(tok::kw_const);
  }

  while (Type->isPointerType()) {
    ParameterTypeQualifiers.push_back(tok::star);
    Type = Type->getPointeeType();

    if (Type.isConstQualified()) {
      ParameterTypeQualifiers.push_back(tok::kw_const);
    }
  }

  std::reverse(ParameterTypeQualifiers.begin(), ParameterTypeQualifiers.end());

  return {Type, ParameterTypeQualifiers, ParameterQualifiers};
}

} // namespace
} // namespace clangd
} // namespace clang
```

Abbreviate Function Template Refactoring — AbbreviateFunctionTemplateTests.cpp

```
//===-- AbbreviateFunctionTemplateTests.cpp ---------------------*- C++ -*-===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//

#include "TweakTesting.h"
#include "gtest/gtest.h"

namespace clang {
namespace clangd {
namespace {

TWEAK_TEST(AbbreviateFunctionTemplate);

TEST_F(AbbreviateFunctionTemplateTest, Test) {
  Header = R"cpp(
      template <typename T>
```

```
    concept foo = true;

    template <typename T>
    concept bar = true;

    template <typename T, typename U>
    concept baz = true;

    template <typename T>
    class list;
)cpp";

ExtraArgs = {"-std=c++20"};

EXPECT_EQ(apply("template <typename T> auto ^fun(T param) {}"),
          " auto fun( auto param) {}");
EXPECT_EQ(apply("template <foo T> auto ^fun(T param) {}"),
          " auto fun( foo auto param) {}");
EXPECT_EQ(apply("template <foo T> auto ^fun(T) {}"),
          " auto fun( foo auto ) {}");
EXPECT_EQ(apply("template <foo T> auto ^fun(T[]) {}"),
          " auto fun( foo auto  [ ]) {}");
EXPECT_EQ(apply("template <foo T> auto ^fun(T const * param[]) {}"),
          " auto fun( foo auto const * param [ ]) {}");
EXPECT_EQ(apply("template <baz<int> T> auto ^fun(T param) {}"),
          " auto fun( baz <int> auto param) {}");
EXPECT_EQ(apply("template <foo T, bar U> auto ^fun(T param1, U param2) {}"),
          " auto fun( foo auto param1,  bar auto param2) {}");
EXPECT_EQ(apply("template <foo T> auto ^fun(T const ** param) {}"),
          " auto fun( foo auto const * * param) {}");
EXPECT_EQ(apply("template <typename...ArgTypes> auto ^fun(ArgTypes...params) "
                "-> void{}"),
          " auto fun( auto ... params) -> void{}");

EXPECT_AVAILABLE("temp^l^ate <type^name ^T> au^to fu^n^(^T par^am) {}");
EXPECT_AVAILABLE("t^emplat^e <fo^o ^T> aut^o fu^n^(^T ^para^m) -> void {}");
EXPECT_AVAILABLE(
    "^templa^te <f^oo T^> a^uto ^fun(^T const ** para^m) -> void {}");
EXPECT_AVAILABLE("templa^te <type^name...ArgTypes> auto "
                 "fu^n(ArgTy^pes...^para^ms) -> void{}");

EXPECT_UNAVAILABLE(
    "templ^ate<typenam^e T> auto f^u^n(list<T> pa^ram) -> void {}");

// Template parameters need to be in the same order as the function parameters
EXPECT_UNAVAILABLE(
    "tem^plate<type^name ^T, typen^ame ^U> auto f^un(^U, ^T) -> void {}");
```

```

  // Template parameter type can't be used within the function body
  EXPECT_UNAVAILABLE("templ^ate<cl^ass T>"
                     "aut^o fu^n(T param) -> v^oid { T bar; }");
}

} // namespace
} // namespace clangd
} // namespace clang
```

Test Project — InlineConceptRequirement.cxx

```
#include <concepts>
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

template<typename T>
concept Foo = requires(T a) {
  { a.abc() } -> std::same_as<int>;
};

// ********************************************
// * Standard case
// ********************************************

// BEFORE:
template <typename  T>
void f() requires std::integral<T>
{}

// AFTER:
// template <std::integral T>
// void f(T) {}

// *******************************************

// BEFORE
template<typename T>
void bar(T a) requires Foo<T> {
  a.abc();
}
```

```
// AFTER
// void f(std::integral<T> auto x) {}

// *******************************************
// * Requires before function
// *******************************************

// BEFORE
template<typename T>
requires std::integral<T>
void f(T) {}

// AFTER
// template <std::integral T>
// void f(T) {}

// *******************************************
// * Example with template template parameter
// *******************************************

// BEFORE
template <template <typename> class Foo, typename T>
void f2() requires std::integral<T>
{}

// *******************************************
// * [NOT SUPPORTED] Multiple requires clauses
// *******************************************

// BEFORE
template <typename T>
void doubleCheck(T) requires std::integral<T> && std::floating_point<T>
{}

// *******************************************
// * [NOT SUPPORTED] Other cases
// *******************************************

// Non-Function Template
template <typename T>
concept FooB = requires(T x) {
    {x} -> std::convertible_to<int>;
};

// Template Template parameters
template <template <typename> class Container, typename T>
concept ContainerWithAllocator = requires {
    typename Container<T>;
};

// Example with multiple requires clauses => no conversion possible
template <typename T>
requires std::integral<T> || std::floating_point<T>
constexpr double Average(std::vector<T> const &vec){
  const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
  return sum / vec.size();
}

int main() {
  int number;

  cout << "Enter an integer: ";
  cin >> number;

  if (number > 5) {
    cout << "Foo ";
  } else {
    cout << "Bar ";
  }

  cout << "You entered " << number;

  cout << "Enter some integers to calculate the average (stop input by typing
any char):";
  std::vector<int> ints;
  while (cin >> number)
    ints.push_back(number);

  cout << "Your average: " << Average(ints);

  return 0;
}
```

Test Project — AbbreviateFunctionTemplate.cxx

```
#include <concepts>
#include <list>
#include <vector>
#include <iostream>
#include <type_traits>
```

```cpp
using namespace std;

// ********************************************
// * Standard case
// ********************************************

// BEFORE
template <typename T>
auto foo1(T param) {}

// AFTER
auto foo2(auto param) {}

// *******************************************

// BEFORE
template <std::integral T>
auto foo3(T param) -> void {}

// AFTER
auto foo4(std::integral auto param) -> void {}

// ********************************************
// * Templated concept as parameter
// ********************************************

// BEFORE
template <std::convertible_to<int> T>
auto foo5(T param) -> void {}

// AFTER
auto foo6(std::convertible_to<int> auto param) -> void {}

// ********************************************
// * Multiple parameters
// ********************************************

// BEFORE
template <typename T, typename U>
auto foo7(T param, U param2) -> void {}

// AFTER
auto foo8(auto param, auto param2) -> void {}

// *******************************************
```

```cpp
// BEFORE
template <typename T, std::integral U>
auto foo9(T param, U param2) -> void {}

// AFTER
auto foo10(auto param, auto param2) -> void {}

// ********************************************
// * Aggregates
// ********************************************

// BEFORE
template <typename...T>
auto fooVar(T...params) -> void{}

// AFTER
auto fooVarTerse(auto ...params) -> void{}

// ********************************************
// * Pointer of a pointer
// ********************************************

// BEFORE
template<std::integral T>
auto foo11(T const ** Tpl) -> void {}

// AFTER
auto foo12(std::integral auto const ** Values) -> void {}

// ********************************************
// * [Not Possible] Collections
// ********************************************

// The keyword `auto` can't be used within containers
template<typename T>
auto foo40(vector<T> param) -> void {}

template<typename T>
auto foo41(list<T> param) -> void {}

template<class T, size_t N>
auto foo42(T (&a)[N], int size) -> void {}

// Using template declaration multiple times for function parameters
template<std::integral T>
auto foo43(T param, T anotherParam) -> void {}
```

```cpp
// Template type parameter is used within the function body
template<std::integral T>
auto foo44(T param) -> void {
    if constexpr (std::is_unsigned_v<T>) {
        std::cout << "The type is an unsigned integer." << std::endl;
    } else {
        std::cout << "The type is not an unsigned integer." << std::endl;
    }
}

// Order in template definition different then the function parameters
// destroys calling of the function
// e.g.: foo14<string, int)(2, "hi");
// when making both param auto the order of the types in `<>` changes!!
template <typename T, std::integral U>
auto foo45(U param, T param2) -> void {}

// Requires clause
template <typename T>
requires std::integral<T>
auto foo46(T param) {}

template <typename T>
auto foo47(T param) requires std::integral<T> {}
```