0.1 Der LSP-Server

Der LSP-Server besteht aus 3 Bereichen:

1. Dem Modell für das LSP-Protokoll.
2. Der Implementierung der Server-Schicht, welche die Kommunikation mit den Clients verwaltet und die Services aufruft.
3. Die Service-Schicht, welche Funktionen des LSP-Protokolls in Service implementiert.

In den folgenden Abschnitten werden der Aufbau, die Aufrufe und die Funktionen dieser Schichten erläutert.

0.1.1 Das LSP-Protokoll

0.1.2 Die Server Implementierung

Abstraktion der Server-Client-Verbindung

Da das LSP-Protokoll Medium unabhängig ist, muss der Server eine Abstraktion für die Verbindung bereitstellen.

```
1  package connect
2  type Connection interface {
3      // WriteMessage Writes Message. May queue the Message the syncronise the Wri
4      WriteMessage(message protokoll.Message)
5
6      // WaitForMessage Waits for the next Message and returns the Message or the
7      // Call to this method blocks execution.
8      WaitForMessage() (protokoll.Message, error)
9
10     BlockResponse(id json.RawMessage)
11     Close() error
12 }
```

Das Interface Connection stellt diese Abstraktion bereit. Es wird für jeden Verbindungstyp implementiert.

0.1.3 Die LSP-Services

Die Schnittstelle für alle Services zu der Server-Schicht bildet das MethodHandler-Interface:

```
1  package service
2  type MethodHandler interface {
3      Initialize(params *initialize.InitializeParams,
4                 result *initialize.InitializeResult)
5      GetMethods() []string
6      HandleMethod(message protokoll.Message)
```

```
7 }
```

Jeder Service implementiert die drei Methoden.

1. **Initialize**
   Der Service liest die Fähigkeiten des Clients und konfiguriert sich selbständig. Sollte der Client den Service nicht unterstützen, muss er sich deaktivieren. Der Service schreibt seine Fähigkeiten in die Antwort des Servers.
2. **GetMethods**
   Gibt die Methoden aus dem LSP-Protokoll zurück für die der Service Meldungen verarbeitet.
3. **HandleMethod**
   Verarbeite die Nachricht.

FileService

Der FileService ist die Schnittstelle zwischen den Dateien, den Parsern und den restlichen Services. Wenn ein anderer Service auf Dateien, semantische Modell oder den Lookup zugreifen möchte, werden Methoden des FileServices genutzt.

```
1 package fileService
2 type FileService struct {
3     handleMap map[string]*fileHandle
4     listeners []FileChangeListener
5     con       connect.Connection
6 }
```

In der “handleMap” werden “fileHandles” gespeichert. Ein FileHandle speichert alle Daten zu einer Datei und wird mit jeder Veränderung aktualisiert.

```
1 package fileService
2 type fileHandle struct {
3     FileContent string
4     Ast         *tree_sitter.Tree
5     Model       *smodel.Model
6     LookUp      *smodel.TypeLookUp
7     Version     int32
8 }
```

Um einen FileHandle zu erzeugen, wird der Inhalt der Datei mithilfe Semantik-Schicht geparst.
Wird der Dateiinhalt geändert, so werden die Änderungen an die Semantik-Schicht übergeben. Dort werden die Änderungen zum iterativen Parsen des neuen Dateiinhalts genutzt. Abschließend werden der Lookup erzeugt und die semantischen Regeln durchlaufen.

FileChangeListener Es gibt Funktionen im LSP-Protokoll die nicht durch eingehende Nachrichten ausgelöst werden, sondern nach Dateiänderungen automatisch an den Client übermittelt werden. Dafür gibt es im DMF-LSP-Server die FileChangeListener.

```
1  package fileService
2  type FileChangeListener interface {
3      // HandleFileChange gets called when the FileService
4      // finishes parsing the File.
5      // It may be called in its own routine.
6      // Changes to the Parameters are ignored.
7      HandleFileChange(file protokoll.DocumentURI, fileContent string,
8                       ast *tree_sitter.Tree, model *smodel.Model,
9                       lookup smodel.TypeLookUp,
10                      errorElements []errElement.ErrorElement,
11                      version int32)
12 }
```

Der FileService enthält Referenzen zu allen aktiven Listener. Nachdem ein FileHandle erstellt oder bearbeitet wurde, werden alle Listener durchlaufen.
Im DMF-LSP-Server ist nur ein FileChangeListener implementiert:

Der DiagnosticsService übermittelt die aktuellen Fehler in der Modelldatei an den Client. Dafür werden alle ErrorElemente in die Diagnostic Strukturen des LSP-Protokolls übersetzt. Schließlich werden die Daten mithilfe einer Request-Nachricht für die Methode “textDocument/publishDiagnostics” an den Client übermittelt.

SemanticTokensService

Um ein schnelles Verständnis einer Datei zu ermöglichen, ist die Einfärbung der Syntax und Semantik wichtig. Dafür stellt das LSP-Protokoll die Möglichkeit bereit semantische Tokens zu übermitteln.
Ein Token bezieht sich immer auf einen Bereich im Sourcecode und übermittelt einen Tokentyp und eine Auswahl der Tokenmodifikatoren. Die Tokentypen und Tokenmodifikatoren werden während der Initialisierung übermittelt. Der Client kann auch während der Initialisierung angeben, ob überlappende Bereich für Token unterstützt werden. Wenn die neuen semantischen Tokens übermittelt werden, werden die Tokens mithilfe einer Zahlenfolge codiert.

Index in der Zahlenfolge für Token mit Index i	Name	Erklärung
5*i	deltaLine	Die Zeilen zwischen dem letzten Token und diesem Token.
5*i+1	deltaStart	Die Zeichen zwischen dem letzen Token und diesem Token. Relativ zu 0, falls der aktuelle Token in einer neuen Zeile ist.
5*i+2	length	Die Länge des Tokens.
5*i+3	tokenType	Index des Typs des Tokens in der Semantik Token Typ Legende.
5*i+4	tokenModifiers	Zahl deren Bits als Wahrheitswerte für jeden Modifikator aus der Legende der Semantik Token Modifikatoren. Der erste Bit (0b00000001) steht dabei für den ersten Modifikator.

Der Legende für Semantik Token Typen

Index	Token Typ	Verwendung im DMF
0	namespace	Für den Namen eines Packages.
1	type	Für das AST-Element “reftype”. Für den Override Wert des Java Typens.
2	class	Für den Override Wert der Java Klasse, der Oberklasse und der implementierten Interfaces.
3	enum	Für den Namen eines Enums.
4	interface	Für den Namen eines Interfaces.
5	struct	Für den Namen eines Structs oder einer Entity.
6	parameter	Für die Parameter einer Funktion.
7	variable	Für die Namen von Argumenten, Referenzen und MultiReferenzen. Für die Namen der Variablen im Entity Identifier. Für den Override Wert des Java Namens.
8	property	
9	number	Für alle Zahlenwerte in einer Enum-Konstante.
10	enumMember	Für den Namen einer Enum-Konstante.
11	function	Für den Namen einer Funktion.
12	comment	Für alle Kommentare.
13	keyword	Für alle Keywords.
14	string	Für alle Strings außer den Werten im Override.
15	modifier	
16	decorator	Für den Override Wert der Java Annotations.

Generierung der Semantik Tokens
Die Semantik Tokens werden mithilfe zweier Algorithmen generiert.

Mithilfe des ASTs werden alle AST-Elemente durchlaufen. Werden Elemente erreicht, deren semantischer Token allein am AST Element bestimmt werden kann, so werden die Token generiert.

Mithilfe des semantischen Modells werden die restlichen Token bestimmt. Dies ist möglich da die Elemente im semantischen Modell Referenzen zum AST beinhalten. Dazu gehören die Namen der verschiedenen Inhalte, wie z.B. von Argumenten. Bei diesen Namen wird das gleiche AST-Element genutzt, wodurch sie nur das semantische Parsen unterscheidbar sind.

Die Codierung der semantischen Tokens muss zunächst die generierten Tokens sortieren, da keine Garantie für die richtige Reihenfolge durch die beiden Algorithmen existiert.

```
1 package semantictokensService
2 slices.SortFunc(semanticElements, func(a, b *semanticElement) int {
3     if a.line == b.line {
4         return cmp.Compare(a.start, b.start)
5     }
6     return cmp.Compare(a.line, b.line)
7 })
```

Nach der Sortierung können die Tokens durchlaufen werden.

```
1  package semantictokensService
2
3  data := make([]uint32, len(semanticElements)*5)
4
5  lastLine := uint32(0)
6  lastStart := uint32(0)
7
8  for i, element := range semanticElements {
9      line := element.line
10     start := element.start
11     length := element.length
12     tokenType := element.tokenType
13     tokenModifiers := element.tokenModifiers
14     if line < lastLine {
15         // Tokens must be sorted by line and character
16         continue
17     }
18
19     if line == lastLine && start < lastStart {
```

```
20          // Tokens must be sorted by line and character
21          continue
22      }
23
24      // Calculate delta encoding
25      deltaLine := line - lastLine
26      deltaStart := uint32(0)
27      if line == lastLine {
28          deltaStart = start - lastStart
29      } else {
30          deltaStart = start
31      }
32
33      data[i*5] = deltaLine
34      data[i*5+1] = deltaStart
35      data[i*5+2] = length
36      data[i*5+3] = tokenType
37      data[i*5+4] = tokenModifiers
38
39      lastLine = line
40      lastStart = start
41  }
```

Dabei wird ein Integer-Slice erstellt. Für jeden Token werden die 5 Zahlen nach dem Protokoll hinzugefügt. Überschneiden sich Token oder sind nicht in der richtigen Reihenfolge werden sie ignoriert. Die aktuelle Zeile und Spalte im Text wird nach jedem Token aktualisiert.