0.1 Der LSP-Server

Der LSP-Server besteht aus 3 Bereichen:

1. Dem Modell für das Language Server 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 LSPs in Services implementiert.

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

0.1.1 Das Language Server Protokoll (LSP)

Das LSP wurde von Microsoft für die Verwendung in Visual Studio Code entwickelt. Es ermöglicht die Funktionen von Plugins über ein Protokoll zu transportieren.
Das Protokoll nutzt JSON-RPC. Wie Nachrichten übermittelt werden, wird im Abschnitt Server Implementierung beschrieben.

Grundlegende Nachrichten

Im LSP basieren alle Nachrichten auf dem Message-Interface. Deshalb beinhaltet jede Nachricht die JSON-RPC-Version auf der sie basiert.

```
1 interface Message {
2     jsonrpc: string;
3 }
```

Die Nachrichten gehören zusätzlich zu einem der folgenden Typen:

Anfragen Request-Message

```
1  interface RequestMessage extends Message {
2
3      /**
4       * The request id.
5       */
6      id: integer | string;
7
8      /**
9       * The method to be invoked.
10      */
11     method: string;
12
13     /**
14      * The method's params.
```

```
15        */
16       params?: array | object;
17   }
```

Eine Anfrage wird genutzt, um das Ergebnis einer Methode anzufragen.
Der Type der Parameter wird von der Methode festgelegt. Die Antwort wird zu der Anfrage mithilfe der ID zugeordnet. Diese ID wird von jedem Teilnehmer hochgezählt.

Antwort Response-Message

```
1  interface ResponseMessage extends Message {
2      /**
3       * The request id.
4       */
5      id: integer | string | null;
6
7      /**
8       * The result of a request. This member is REQUIRED on success.
9       * This member MUST NOT exist if there was an error invoking the metho
10      */
11     result?: LSPAny;
12
13     /**
14      * The error object in case a request fails.
15      */
16     error?: ResponseError;
17 }
```

Antworten werden immer nach einer Anfrage geschickt, selbst bei einem Fehler.
Die meisten LSP-Clients geben dem Server für die Antwort 5 Sekunden Zeit.
Die ID beinhaltet die ID aus der Anfrage.
Das Ergebnis('result') existiert immer bei einer erfolgreichen Durchführung der Methode. Soll kein Wert zurückgegeben werden, so wird beim Ergebnis der Wert "null" gesetzt.
Bei einem Fehler während der Ausführung der Methode muss der Fehler('error') in der Antwort befüllt werden.

```
1  interface ResponseError {
2      /**
3       * A number indicating the error type that occurred.
4       */
5      code: integer;
6
7      /**
8       * A string providing a short description of the error.
9       */
10     message: string;
11
12     /**
13      * A primitive or structured value that contains additional
```

```
14       * information about the error. Can be omitted.
15       */
16      data?: LSPAny;
17  }
```

Der beinhaltet zwei wichtige Elemente: den Error-Code und die Error-Nachricht. Die Error-Codes sind vordefiniert vom Protokoll.[**response**] Die Nachricht ist frei wählbar, sollte jedoch dem Nutzer erklären, wodurch der Fehler entstand.
Es können zusätzliche Daten übermittelt werden. Jedoch ist das Format nicht definiert. Es bietet sich nur an dieses Element zu nutzen, wenn sowohl Server als auch Client selbst implementiert wurden.

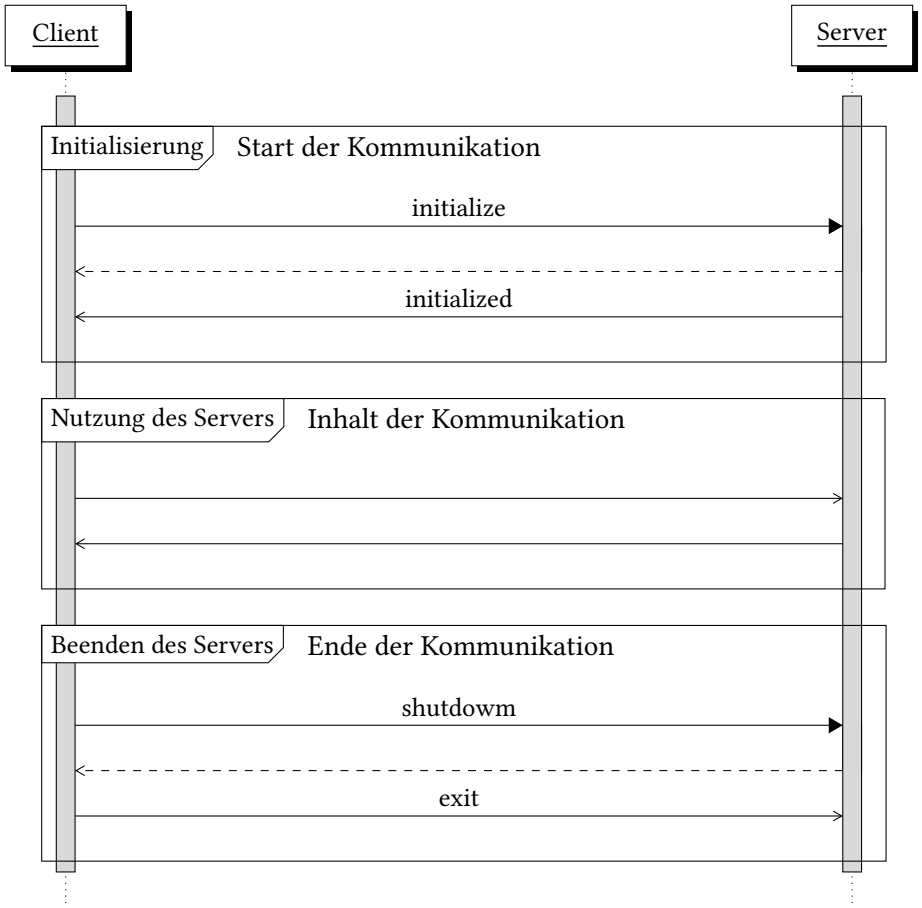
Benachrichtigungen Notification-Message

```
1  interface NotificationMessage extends Message {
2      /**
3       * The method to be invoked.
4       */
5      method: string;
6
7      /**
8       * The notification's params.
9       */
10     params?: array | object;
11 }
```

Benachrichtigungen übermitteln Daten, ohne eine Antwort zu erwarten. Sie werden für z.B. den Abbruch einer Anfrage oder dem Übermitteln der aktuellen Fehler genutzt.
Das Datenmodell einer Benachrichtigung unterscheidet sich von einer Anfrage nur durch die fehlende ID.

Der LSP-Kommunikation-Lebenszyklus

Die Kommunikation eines Clients mit einem Server folgt einem Ablauf, welcher den Anfang und das Ende der Kommunikation definiert. Der Start eines Vorgangs im Lebenszyklus wird immer vom Client gestartet.



Initialisierung

Die Initialisierung umfasst die ersten Nachrichten, nachdem die Transportschicht die Verbindung etabliert hat.
Sie beginnt mit dem Aufruf der Methode ‘initialize’ des Servers. In den Parametern des Aufrufs werden die vom Client unterstützten Funktionen des LSPs übermittelt. Der Server antwortet mit der Auswahl aus den vom Client gesendeten Funktionen, die auch der Server unterstützt. Die Unterstützung einer Funktion muss keine Entscheidungsfrage sein. Server und Client übermitteln auch wie sie Funktionen unterstützen. So können Client oder Server kommunizieren, dass sie z.B. die Veränderungen an einer Datei und/oder die vollständigen Dateien unterstützen. (Für genauere Informationen siehe Abschnitt Dokumenten Synchronisation)

Nachdem die Feinheiten der Kommunikation bekannt sind, beginnt der Server mit den Vorbereitungen, um alle vereinbarten Funktionen verarbeiten zu können. Ist diese Vorbereitung abgeschlossen, sendet der Server die ‘initialized’ Benachrichtigung. Nun ist der Start der Kommunikation ist erfolgreich abgeschlossen.

Beenden des Servers

Wird der LSP-Server nicht mehr vom Client benötigt, z.B. da keine relevanten Dateien mehr geöffnet sind oder die IDE/das Projekt geschlossen wird, so wird die Methode ‘shutdown’ aufgerufen. Die Methode erwartet leere Parameter und beendet alle Strukturen für die Verarbeitung der LSP-Funktionen.

Sobald der Client die Kommunikation beendet, sendet er eine ‘exit’-Benachrichtigung. Hat der Server vorher eine ‘shutdown’-Anfrage erfolgreich beantwortet, so soll er mit dem Exit-Code 0 beenden. Ansonsten soll er sich mit dem Exit-Code 1 beenden.
Der Exit-Code kann nur vom Client gelesen werden, wenn dieser den Prozess des Servers auch gestartet hat.

In den folgenden Abschnitten werden die Funktionen die der DMF-LSP-Server unterstützt vorgestellt.

Anfragen stornieren

Cancelation Support ($/cancelRequest) [**cancelation**]

Der Client kann jederzeit entscheiden, dass das Ergebnis einer Anfrage nicht mehr benötigt wird.

Definition der Parameter [**cancelation**]

```
interface CancelParams {
    /**
     * The request id to cancel.
     */
    id: integer | string;
}
```

Die Parameter enthalten die ID der Anfrage, dessen Ergebnis nicht mehr benötigt wird. Der Server muss jedoch die Anfrage trotzdem noch beantworten, um dem Ablauf des Protokolls zu folgen.

Diese Funktion wird im subsubsec:cancel-service implementiert.

Dokumenten Synchronisation

Text Document Synchronization
(textDocument/didOpen, textDocument/didChange, textDocument/didClose) [**dokumente**]

Damit ein LSP-Server Aussagen über eine Datei oder ein ganzes Projekt treffen kann, muss der Server den aktuellen Stand der Dateien kennen.

Während der Initialisierung wird vom Server festgelegt, ob das Öffnen und Schließen von Dateien an den Server übertragen und wie die Bearbeitungen an den Server übertragen werden sollen (gar nicht/komplette Datei/inkrementell). Der DMF-LSP-Server nutzt sowohl das Öffnen und Schließen der Dateien als auch die inkrementelle Übertragung der Bearbeitungen.

Die Synchronisation beginnt, mit der Methode ‘textDocument/didOpen’, welche beim Öffnen einer Datei ausgeführt wird.

Definition der Parameter [**dokumente**]

```
interface DidOpenTextDocumentParams {
    /**
     * The document that was opened.
     */
    textDocument: TextDocumentItem;
}
interface TextDocumentItem {
    /**
     * The text document's URI.
     */
    uri: DocumentUri;

```

```
13      /**
14       * The text document's language identifier.
15       */
16      languageId: string;
17
18      /**
19       * The version number of this document (it will increase after each
20       * change, including undo/redo).
21       */
22      version: integer;
23
24      /**
25       * The content of the opened text document.
26       */
27      text: string;
28  }
```

Die Parameter der Methode enthalten die Uniform Resource Identifier (URI), den Inhalt der Datei, die Sprache und die Version. Von diesen Parametern sind die URI und der Inhalt sehr bedeutend. Sie bestimmen welcher Inhalt unter welcher URI vom Server gespeichert und verarbeitet wird.

Nachdem ein Dokument geöffnet wurde, wird bei jeder Änderung die Methode 'textDocument/didChange' aufgerufen. Die Parameter enthalten neben der URI die Änderungen, welche die direkt in die Bearbeitungen der Treesitter-API übersetzt werden können.

Definition der Bearbeitungen (ohne irrelevante Elemente)[**dokumente**]

```
1  export type TextDocumentContentChangeEvent = {
2      /**
3       * The range of the document that changed.
4       */
5      range: Range;
6
7      /**
8       * The new text for the provided range.
9       */
10     text: string;
11 }
```

Wird eine Datei geschlossen, können im Server alle Ressourcen für die Datei freigegeben werden. Dafür ruft der Client die Methode 'textDocument/didClose' auf.

Die Funktionen für die Dokumenten Synchronization werden im FileService implementiert.

Referenzen bestimmen

Go to Declaration & Find References

(textDocument/declaration, textDocument/references) [**declaration**] [**references**]

Referenzen sind ein großer Teil des Typsystems des DMFs. Damit diese Referenzen auch nachvollziehbar für die Entwickler*innen sind, bietet das LSP mehrere Funktionen an.

Die Deklaration eines Typs kann mit der Methode 'textDocument/declaration' abfragt werden.

Listing 1: Definition der Deklarations-Parameter [**declaration**]

```
export interface DeclarationParams extends TextDocumentPositionParams,
    WorkDoneProgressParams, PartialResultParams {
}
interface TextDocumentPositionParams {
    /**
     * The text document.
     */
    textDocument: TextDocumentIdentifier;

    /**
     * The position inside the text document.
     */
    position: Position;
}
```

Die Deklaration wird als LocationLink als Ergebnis bereitgestellt.

Definition des LocationLink [**declaration**]

```
interface LocationLink {

    /**
     * Span of the origin of this link.
     *
     * Used as the underlined span for mouse interaction. Defaults to the wor
     * range at the mouse position.
     */
    originSelectionRange?: Range;

    /**
     * The target resource identifier of this link.
     */
    targetUri: DocumentUri;

    /**
     * The full target range of this link. If the target for example is a sym
     * then target range is the range enclosing this symbol not including
     * leading/trailing whitespace but everything else like comments. This
     * information is typically used to highlight the range in the editor.
     */
    targetRange: Range;

    /**
     * The range that should be selected and revealed when this link is being
     * followed, e.g the name of a function. Must be contained by the
     * `targetRange`. See also `DocumentSymbol#range`
     */
```

```
29      targetSelectionRange: Range;
30  }
```

Ein LocationLink beschriebt einen Bereich in einem, auch vom aktuellem Dokument unterschiedlichem, Dokument. Dabei wird zwischen dem kompletten Bereich der Deklaration und dem Bereich welcher automatisch ausgewählt und in einer Auswahl angezeigt werden soll.

Alle Referenzen zu einem Typ können mit der Methode 'textDocument/references' abgefragt werden. Zu den Referenzen gehören die Deklarationen und die Verwendung des Typs in Referenzen, Multireferenzen, Funktionen und Abstraktionen.
Die Parameter unterscheiden sich von den der Deklaration nur im ReferenceContext. Dieser beinhaltet die Information, ob die Deklaration in der Antwort enthalten sein soll.

Definition des ReferenceContext [**references**]

```
1  export interface ReferenceContext {
2      /**
3       * Include the declaration of the current symbol.
4       */
5      includeDeclaration: boolean;
6  }
```

Im Ergebnis werden die Referenzen nicht in einem LocationLink zurückgegeben, sondern nur in einer Location. Diese Location enthält nur die URI der Datei und den Bereich der Referenz.

Definition der Location [**references**]

```
1  interface Location {
2      uri: DocumentUri;
3      range: Range;
4  }
```

Die Beschreibung der Implementierung beider Methoden befindet sich im Abschnitt ReferenceService.

Hover-Effekt

Hover (textDocument/hover) [**hover**]

Das LSP bietet die Funktion Informationen über ein Element bereitzustellen, wenn die Entwickler*innen über den Text "hovern".

Während der Initialisierung gibt der Client die Formate an, die er für die Dokumentation unterstützt. Das LSP beinhaltet zwei Formate in der Spezifikation: normaler Text und Mardown.

Die Parameter der Anfrage erben von den TextDocumentPositionParams (siehe Definition der Deklarations-Parameter [**declaration**]). Sie enthalten die URI der Datei und die Position.
Die Dokumentation wird zusammen mit einem optionalen Bereich übermittelt.

Listing 2: Definition des Hover Ergebnis [**hover**]

```
1  /**
2   * The result of a hover request.
3   */
```

```
4  export interface Hover {
5      /**
6       * The hover's content
7       */
8      contents: MarkedString | MarkedString[] | MarkupContent;
9
10     /**
11      * An optional range is a range inside a text document
12      * that is used to visualize a hover, e.g. by changing the background col
13      */
14     range?: Range;
15 }
16 export interface MarkupContent {
17     /**
18      * The type of the Markup
19      */
20     kind: MarkupKind;
21
22     /**
23      * The content itself
24      */
25     value: string;
26 }
27 export type MarkupKind = 'plaintext' | 'markdown';
```

Die Implementierung des Hover-Effekts wird im Abschnitt HoverService beschrieben.

Faltbereich

Folding Ranges (textDocument/foldingRange) [**folding**]

Die Möglichkeit den Code in Abschnitte zu unterteilen und diese einfalten zu können, erleichtert die Übersicht in großen Dateien. Deshalb definiert das LSP eine Funktion, um diese Bereiche an die IDE zu übermitteln.

Während der Initialisierung kann der Client viele Vorgaben und Wünsche an den Server machen. Dazu zählen die gewünschte maximale Anzahl der Bereiche, ob nur komplette Zeilen gefaltet werden können, welche Falttypen unterstützt werden und ob vom Server generierte Zusammenfassungen angezeigt werden können.

Die Anfrage an den Server beinhaltet nur die URI der Datei.
Die Antwort des Servers enthält eine Liste mit Faltbereichen. Die Faltbereiche decken die komplette Datei ab. Jeder Faltbereich enthält zusätzlich zur Startposition und Endposition auch den Falttypen und optional auch eine Zusammenfassung.

Die Implementierung der Faltbereiche wird im Abschnitt Faltbereich beschrieben.

Auswahlbereich

Selection Range (textDocument/selectionRange) [**selection**]

Durch die unterschiedlichen Grammatiken aller Programmiersprachen ist eine Verallgemeinerung der Auswahlbereiche in einem Dokument unmöglich. Deshalb bietet das LSP die Möglichkeit diese Bereiche vom Server abzufragen.

Die Anfrage beinhaltet ein Dokument und verschiedene Positionen, zu denen die Auswahlbereiche erfragt werden. Der Vorteil von mehreren Positionen ist die Bündlung der Anfragen für Editoren mit mehreren Eingabemarken(Cursor).

Auszug aus den Parametern

```
type SelectionRangeParams struct {
    // TextDocument identifies the document to compute selection ranges fo
    TextDocument protokoll.TextDocumentIdentifier `json:"textDocument"`

    // Positions is an array of positions in the text document for which t
    // selection ranges.
    Positions []protokoll.Position `json:"positions"`
}
```

Auswahlbereiche bilden einen Auszug aus dem AST. Um eine Auswahl zu vergrößern oder zu verkleinern wird der höhere bzw. tiefere Auswahlbereich aus der Hierarchie des ASTs benötigt. Deshalb beinhaltet die Antwort im LSP auch die Möglichkeit pro Position eine Kette an Auswahlbereichen zu liefern. Diese Kette wird durch das Parent-Attribut gebildet.

Antwort des Servers

```
// SelectionRange represents a selection range with its parent selection r
type SelectionRange struct {
    // Range is the actual range of this selection range.
    Range protokoll.Range `json:"range"`

    // Parent is the parent selection range containing this range. Therefor
    // multiple selection ranges can be encoded into a tree structure.
    Parent *SelectionRange `json:"parent,omitempty"`
}

// SelectionRangeResult represents the result of a selection range request
// It's an array of SelectionRange objects, one for each position in the r
type SelectionRangeResult []SelectionRange
```

Die Implementierung der Auswahlbereiche wird im Abschnitt SelectionRangeService beschrieben.

Semantische Tokens

Semantic Tokens (textDocument/semanticTokens/full) [**semantic**]

Um ein schnelles Verständnis einer Datei zu ermöglichen, ist die Einfärbung der Syntax und Semantik wichtig. Dafür stellt das LSP die Möglichkeit bereit semantische Tokens/Symbole 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.

Ein Client kann Anfragen für die semantischen Token an den Server stellen. Für das DMF wurde nur die Methode ‘textDocument/semanticTokens/full’ welche alle semantischen Token für eine Datei generiert beachtet.

Die Codierung des semantischen Token

Wenn die neuen semantischen Tokens übermittelt werden, werden die Tokens mithilfe einer Zahlenfolge codiert. Dies führt zu einer starken Komprimierung des Ergebnisses, welche besonders relevant für diese Methode des LSPs ist, da das Ergebnis besonders viele Daten beinhaltet.
Die Einträge der Zahlenfolge werden nach dem folgenden Schema 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 letzten 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.

Die Implementierung der semantischen Tokens wird im Abschnitt SemanticTokensService Protokoll in Abschnitt Semantische Tokens beschrieben.

Diagnosen

Publish Diagnostics (textDocument/publishDiagnostics) [**diagnostics**]

Um Fehler, Warnungen, Informationen und Hinweise den Entwickler*innen anzeigen zu können bietet das LSP die Möglichkeit Diagnosen zu übermitteln. Hierbei ist die Besonderheit, dass nur der Server weiß, ob und wann die Diagnosen einer Datei sich verändern. Deshalb bietet das LSP die Möglichkeit, dass der Server Benachrichtigungen an den Client mit den Diagnosen sendet.

Während der Initialisierung kann der Client (neben der Unterstützung von Feinheiten der Spezifikation) angeben, ob er Diagnosen-Tags (eng. diagnostic tag) unterstützt. Diese Tags differenzieren die Diagnosen, wie die semantischen Modifikatoren die semantisch Tokens weiter differenzieren.

Die Benachrichtigung beinhalten eine Liste mit Diagnosen. Jede Diagnose bezieht sich auf einen Bereich im Sourcecode. Eine Diagnose kann ein Fehler, eine Warnung, eine Information oder ein Hinweis sein.
Der Unterschied zwischen einer Information und einem Hinweis liegt Bedeutsamkeit der enthaltenen Information. Eine Diagnose mit einer Information sollte beim Ermitteln eines Fehlers vor den Hinweisen beachtet werden.
Der Inhalt einer Diagnose setzt sich aus den Feldern ‘Message’ und ‘Source’ zusammen. ‘Message’ beschreibt den Fehler und ‘Source’ den Grund.
Es können auch zusätzliche Informationen in einer Diagnose enthalten sein. Diese könne andere Stellen im Code erwähnen, die für die Diagnose bedeutend sind.

Listing 3: Definition des Inhalts der Benachrichtigung

```
// Diagnostic represents a diagnostic, such as a compiler error or warning
type Diagnostic struct {
    // Range is the range at which the message applies
    Range Range `json:"range"`
    // Severity is the diagnostic's severity. Can be omitted. If omitted it
    // client to interpret diagnostics as error, warning, info or hint
    Severity *DiagnosticSeverity `json:"severity,omitempty"`
    // Source is a human-readable string describing the source of this diag
    Source string `json:"source,omitempty"`
    // Message is the diagnostic's message. It usually appears in the user
    Message string `json:"message"`
    // Tags provides additional metadata about the diagnostic
    Tags []DiagnosticTag `json:"tags,omitempty"`
    // RelatedInformation provides related diagnostic information
    RelatedInformation []DiagnosticRelatedInformation `json:"relatedInforma
}
// DiagnosticRelatedInformation represents related diagnostic information,
type DiagnosticRelatedInformation struct {
    // Location of this related diagnostic information
    Location Location `json:"location"`
    // Message is a message about this related diagnostic information
    Message string `json:"message"`
}
```

Die Implementierung für die Erstellung der Diagnosen wird im Abschnitt DiagnosticsService beschrieben.

0.1.2 Server Implementierung

Die Serverschicht abstrahiert die Verbindung zum Client und das JSON-RPC Protkoll. So bekommen die Services direkt Nachrichten und können auch Nachrichten verschicken.

Abstraktion der Server-Client-Verbindung

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

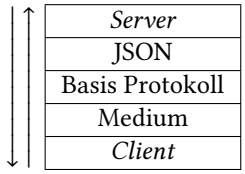
```
package connect
type Connection interface {
    // WriteMessage Writes Message.
    // May queue the Message the syncronise the Writing
    WriteMessage(message protokoll.Message)

    // WaitForMessage Waits for the next Message
    // and returns the Message or the error.
    // Call to this method blocks execution.
    WaitForMessage() (protokoll.Message, error)

    BlockResponse(id json.RawMessage)
```

```
13      Close() error
14  }
```

Das Interface Connection stellt diese Abstraktion bereit. Es wird für jeden Verbindungstyp implementiert. Zwischen den Verbindungstypen unterscheiden sich nur die Implementierungen der Interfaces der Standardbibliothek.
Die Verbindung durchläuft immer die gleichen Schritte, nur die Richtung unterscheidet sich bei Lesen und Schreiben.



JSON

Damit die Nachrichten übermittelt werden können, müssen sie in JSON Text geparsed werden. Dafür wird die Standardbibliothek von Golang genutzt.

Listing 4: Umwandlung von und zu JSON

```
1  message := protokoll.Message{}
2  // Golang -> JSON
3  data, err := json.Marshal(message)
4  // JSON -> Golang
5  err = json.Unmarshal(data, &message)
```

Das Verhalten der verschiedenen Datenstrukturen beim Parsen wird direkt bei der Definition angegeben. Deshalb muss keine zusätzlich Logik für Werte die nicht übertragen werden soll, falls sie leer sind, oder für unterschiedliche Namen im JSON-Text implementiert werden.

Basis Protokoll

Das LSP besitzt nicht Spezifikationen für den Inhalt der Nachrichten, sondern auch für deren Übertragung. Vorrausgesetzt wird nur eine bidirektionale Verbindung, welche parallel Text übertragen kann.
Jede Nachricht im Basis Protokoll besteht aus zwei Teilen: dem Header und dem Inhalt.

Der Header enthält Angabe zum Einlesen. Diese werden jeweils in eine eigene Zeile geschrieben. Die Zeilen werden mit den Kontrollzeichen '\r\n' beendet.

Variable	Beschreibung
Content-Length	Die Länge der übertragenen Nachricht (in Bytes).
Content-Type	Der -Typ der Nachricht. Wenn nicht angegeben: 'application/vscode-jsonrpc; charset=utf-8'

[**header**]

Der Header wird beendet einer leeren Zeile welche dennoch mit beiden Kontrollzeichen beendet wird. Zuletzt folgt der Inhalt der Nachricht. Dabei ist wichtig, dass die angegebene Länge genau der Länge der Nachricht entspricht.

Um das Basis Protokoll Lesen zu können, wird der 'BufReader' der Golang-Standardbibliothek verwendet. Beim Lesen aus dem 'BufReader' kann ein Zeichen bis zu dem gelesen werden soll, angegeben werden. So

kann die Verbindung Zeilenweise ausgelesen werden.
Beginnt eine Zeile mit dem Identifier der Länge der Datei so wird der Rest der Zeile als Zahl interpretiert und zwischengespeichert.
Wird eine leere Zeile gelesen wird die Schleife zum Auslesen des Headers beendet.
Nun wird ein Buffer mit der gespeicherten Größe angelegt und mit dem Inhalt aus der Verbindung befüllt.
Der Inhalt des Buffers kann nun als Text weiter verwendet werden.

Das Schreiben im Basis Protokoll beginnt mit Messung der Nachrichtenlänge. Nun kann der Header in die Verbindung geschrieben werden. Schließenlich wird der Inhalt in die Verbindung geschrieben.

Medium

Der DMF-LSP-Server unterstützt zwei Medien: Standard-IO und TCP-Sockets.
Standard-IO bietet sich für die meisten Fälle an, da der Prozess häufig von der IDE verwaltet wird. Diese Übertragung enthält auch keine weiteren Schichten und ist somit sehr schnell. Sie vereinfacht auch den Server Prozess, da es nur einen Client geben kann.

Listing 5: Start eines StdIO-Servers

```
1 newServer := server.NewServer(connect.NewStdIOConnection())
2 newServer.MessageLoop()
```

Wenn der Prozess nicht von der ide verwaltet werden kann und der Server zentral gehostet werden soll, können TCP-Sockets genutzt werden. Sie ermöglichen mehreren Nutzern eine Server Instanz zu nutzen und die Nutzung eines Servers, welcher nicht auf dem gleichen Computer läuft.
Bei der Implementierung ist die Unterscheidung zwischen dem Begriff Server und der Datenstruktur 'Server' zu beachten. Für jede Verbindung wird eine eigene Server-Instanz in einer eigenen Routine erzeugt.

Listing 6: Verwaltung des TCP-Servers

```
1  listener, err := net.Listen("tcp", args.Port)
2  for {
3      conn, err := listener.Accept()
4
5      // Server starten
6      go func(conn net.Conn) {
7          newServer := server.NewServer(
8                           connect.NewSocketConnection(conn))
9          newServer.MessageLoop()
10     }(conn)
11 }
```

Abstraktion des JSON-RPC-Protokolls

Die Abstraktion des JSON-RPC-Protokolls verbindet die Services und die Verbindung zum Client. Sie beinhaltet zwei Phasen: Intialisierung und Verarbeitung.
Die Intialisierung erstellt alle Services und trägt diese eine Map ein. Die Map enthält die Referenzen zwischen den Methodennamen und den Services. Die Methodennamen liefern die Services, so kann ein Service mit nur zweilen Zeilen hinzugefügt werden:

```
1 newCancelService := cancelService.NewCancelService(con)
2 s.addHandler(newCancelService)
```

Die Verarbeitung...

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 zurück für die der Service Meldungen verarbeitet.
3. **HandleMethod**
   Verarbeite die Nachricht.

FileService

Protokoll in Abschnitt Dokumenten Synchronisation

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 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 (Protokoll im Abschnitt 0.1.1) übermittelt die aktuellen Fehler in der Modelldatei an den Client. Dafür werden alle ErrorElemente in die Diagnostic Strukturen des LSPs übersetzt. Schließlich werden die Daten mithilfe einer Request-Nachricht für die Methode “textDocument/publishDiagnostics” an den Client übermittelt.

SemanticTokensService

Protokoll in Abschnitt Semantische Tokens

Der SematicTokensService implementiert die Methode für die semantischen Tokens. Diese Tokens werden für die Einfärbung des Textes einer Datei nach der Syntax und Semantik genutzt. Mit dieser Einfärbung des Sourcecodes können Entwickler*innen schnell ein Verständnis der Datei entwickeln.

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. Dabei ist für die Codierung (siehe Die Codierung des semantischen Token) der Index in den Listen entscheidend.
Für das DMF wird nur die folgende Auswahl der Tokenmodifikatoren genutzt.

1	declaration
2	definition

Die Tokentypen und ihre Verwendung zusammen mit den Tokenmodifikatoren

Index	Token Typ	Verwendung im DMF Tokenmodifikator(-en) | Verwendung	
0	namespace	declaration	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	declaration	Für den Namen eines Enums.
4	interface	declaration	Für den Namen eines Interfaces.
5	struct	declaration	Für den Namen eines Structs oder einer Entity.
6	parameter	declaration	Für die Parameter einer Funktion.
7	variable	-	Für die Namen der Variablen im Entity Identifier. Für den Override Wert des Java Namens.
		declaration	Für die Namen von Argumenten, Referenzen und Multi-Referenzen.
8	property	Platzhalter	
9	number	definition	Für alle Zahlenwerte in einer Enum-Konstante.
10	enumMember	declaration	Für den Namen einer Enum-Konstante.
11	function	declaration	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	Platzhalter	
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. Dazu gehören die meisten semantischen Token.

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 semantischen Tokens befinden sich nun in einer Liste mit semantischen Elementen. Diese müssen für die Antwort codiert werden.

Listing 7: semantisches Element

```
1 type semanticElement struct {
2     line            uint32
3     start           uint32
4     length          uint32
```

```
5       tokenType       uint32
6       tokenModifiers  uint32
7   }
```

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

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

Nach der Sortierung können die Tokens durchlaufen werden.

```
1   data := make([]uint32, len(semanticElements)*5)
2
3   lastLine := uint32(0)
4   lastStart := uint32(0)
5
6   for i, element := range semanticElements {
7       line := element.line
8       start := element.start
9       length := element.length
10      tokenType := element.tokenType
11      tokenModifiers := element.tokenModifiers
12      if line < lastLine {
13          // Tokens must be sorted by line and character
14          continue
15      }
16
17      if line == lastLine && start < lastStart {
18          // Tokens must be sorted by line and character
19          continue
20      }
21
22      // Calculate delta encoding
23      deltaLine := line - lastLine
24      deltaStart := uint32(0)
25      if line == lastLine {
26          deltaStart = start - lastStart
27      } else {
28          deltaStart = start
29      }
30
31      data[i*5] = deltaLine
32      data[i*5+1] = deltaStart
33      data[i*5+2] = length
```

```
34      data[i*5+3] = tokenType
35      data[i*5+4] = tokenModifiers
36
37      lastLine = line
38      lastStart = start
39  }
```

Beim Durchlaufen 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 die Token ignoriert.
Die aktuelle Zeile und Spalte im Text wird nach jedem Token aktualisiert.

SelectionRangeService

Protokoll in Abschnitt Auswahlbereich

Der SelectionRangeService implementiert die LSP-Methode "textDocument/selectionRange". Diese wird dafür genutzt, dass bei der Auswahl immer der richtige Text ausgewählt wird.
Die Bereiche werden im Service mithilfe des ASTs berechnet.

```
1  package selectionRangeService
2  func (s *SelectionRangeService) computeSelectionRangeForPosition(
3          content fileService.FileContent,
4          position protokoll.Position) *selectionRange.SelectionRange {
5      finder := util.NewNodeFinder([]byte(content.Content))
6      node := finder.FindSmallestNodeAroundPosition(content.Ast, position)
7      var currentRange *selectionRange.SelectionRange
8      for node != nil {
9          newRange := &selectionRange.SelectionRange{
10             Range: protokoll.ToRange(node.Range()),
11         }
12         newRange.Parent = currentRange
13         currentRange = newRange
14
15         node = node.Parent()
16     }
17     return currentRange
18 }
```

Es wird zunächst das kleinste Element(das "Blatt" des Baums) des ASTs für die Position Code bestimmt.
Nun wird durch die Schichten des ASTs bis zur Wurzel iteriert. Bei jedem Element eine neue "SelectionRange" Instanz angelegt. Dieses "SelectionRange" enthält immer den Bereich und eine Referenz zu der vorherigen "SelectionRange". Durch die Referenz wird die Baum-Struktur auch im Ergebnis übermittelt und es müssen weniger Anfragen an den Server gesendet werden.

Ein Request kann mehrere Positionen enthalten. Deshalb wird der Algorithmus für jede Position ausgeführt.

FoldingService

Protokoll in Abschnitt Faltbereich

Der FoldingService stellt die Methode ‘textDocument/foldingRange’, für die Übermittlung der faltbaren Bereiche, bereit. Es werden drei verschiende Bereicharten berechnet: PackageElemente, Kommentare und der Import-Block.

Für die PackageElemente werden die AST-Nodes des Elements durchlaufen. Es wird immer der Bereich, welcher von den geschweiften Klammern umschlossen wird, gefaltet. Deshalb werden die auch diese Nodes beim Durchlaufen gesucht.

Listing 8: Suche nach Start- und Ende-Node des Faltbereichs eines PackageElements

```
1  cursor.GotoFirstChild()
2  for {
3      if cursor.Node().GrammarName() == "{" {
4          startNode = cursor.Node()
5      }
6      if cursor.Node().GrammarName() == "}" {
7          endNode = cursor.Node()
8      }
9      if !cursor.GotoNextSibling() {
10         break
11     }
12 }
```

Mithilfe diese Nodes kann auf die Zeilen und Spalten beider Positonen zugegriffen werden. Damit die Klammern auch noch in der IDE angezeigt werden, wird der Startzeichenindex (Start-Spalte) um 1 erhöht und der Endzeichenindex (End-Spalte) um 1 gesenkt.

Für die Kommentare wird die Query-Funktion von Treesitter genutzt.

```
1  (comment_block) @cb
```

Die Ergebnisse einer Query werden in der Treesitter-API QueryMatches genannt. Für diese Query enthalten sie alle ‘comment_block’ Nodes im AST. Durch die Ergebnisse wird iteriert und für jeden QueryMatch ein Faltbereich erstellt. Dabei wird zwischen einzeiligen und mehrzeiligen Kommentaren unterschieden, die Position unterschiedlich interpretiert werden müssen. Einzeilige Kommentare werden nicht in jeder IDE gut verarbeitet, deshalb können sie über die Argumente ausgeschaltet werden.

Listing 9: Auszug aus der Bestimmung der Faltbereiche für die Kommentare

```
1  queryCursor := tree_sitter.NewQueryCursor()
2  captures := queryCursor.Captures(f.kommentarQuery,
3                                   root.RootNode(), nil)
4  for match, index := captures.Next(); match != nil;
5          match, index = captures.Next() {
6      node := match.Captures[index].Node
7      startPos := node.StartPosition()
8      endPos := node.EndPosition()
9
```

```
10      if startPos.Row != endPos.Row {
11          // Mehrzeilige Kommentare
12      }else if !args.DisableSingleLineCommentsFolding{
13          // Einzeilige Kommentare
14      }
15  }
```

Bei allen Kommentaren liegt das Ende im Ergebnis der Query in der nächsten Zeile. Der Faltbereich soll jeden am Ende der Kommentarzeile enden. Deshalb wird die Endposition verschoben.
Bei mehrzeiligen Kommentaren müssen Startzeichenindex und Endzeichenindex anhand von unterschiedlichen Zeilen bestimmt werden. Dafür muss der Inhalt des Kommentars in Zeilen aufgeteilt werden. Die erste und letzte Zeile werden für die Indexe genutzt. Für die Zusammenfassung wird die
Bei einzieligen Kommentaren könen die Indexe anhand der einen Zeile bestimmt werden.

Für den Import-Block wird die Node des Import-Blocks aus dem AST genutzt. Diese enthält jedoch alle Zeilen, ab dem ersten Import-Statement bis zum ersten PackageElement. Deshalb werden die Leerzeilen am Ende des Import-Blocks herrausgefiltert.

Listing 10: Filterung der leeren Zeilen

```
1  var endPos tree_sitter.Point
2  for {
3      if cursor.Node().GrammarName() == "import_block" {
4          importBlock = cursor.Node()
5          cursor.GotoLastChild()
6          endPos = cursor.Node().EndPosition()
7          text := cursor.Node().Utf8Text([]byte(content))
8          split := strings.Split(text, "\n")
9          length := len(split)
10         for i := range split {
11             s := split[length-i-1]
12             if strings.ContainsFunc(s, func(r rune) bool {
13                 switch r {
14                 case ' ', '\n':
15                     return false
16                 default:
17                     return true
18                 }
19             }) {
20                 endPos.Column = uint(len(split[length-i-1]))
21             } else {
22                 endPos.Row--
23             }
24         }
25         break
26     }
27     if !cursor.GotoNextSibling() {
28         break
29     }
30 }
```

ReferenceService

Protokoll in Abschnitt Referenzen bestimmen

Um zwischen den Referenzen und den Deklarationen in der IDE in der Datei springen zu können, implementiert der ReferenzService die Methoden 'textDocument/references' und 'textDocument/declaration'.
Die Parameter für beide Methoden enthalten das Textdokument und die Position im Dokument. Um die Referenz zu finden, wird der AST vom Blatt an der Position zur Wurzel durchlaufen. Dabei werden für folgende Node-Typen die erste Node gespeichert.

AST-Node Namen	Verwendung
reftype	Referenzen, Multireferenzen oder Parameter
identifier	Namen von Variablen oder Funktionen
package_block, struct_block, enum_block, entity_block, interface_block	PackageElemente

Mithilfe dieser Nodes kann der Name für den LookUp der PackageElemente und der Name des NamedElements innerhalb des PackageElements bestimmt werden.

Referenzen

Nachdem das PackageElement und der Name bestimmt wurden, wird unterschieden ob es sich um einen Typ oder um eine Variable handelt.
Für die **Variablen** wird das aktuelle Element und alle SubElemente (Elemente durch Abstraktion) durchsucht. In den Elementen wird nach der Variable über den NamendElemente-LookUp und in dem EntityIdentifier gesucht.
Für die Suche nach **Typ**-Referenzen müssen alle PackageElemente durchsucht werden. In den PackageElementen werden Referenzen, Multireferenzen, Parameter und die Abstraktion durchsucht.

Deklarationen

Auch bei den Deklarationen wird zwischen Typen und Variablen unterschieden.
Für die **Variablen** wird das Element und die SubElemente durchsucht. Für die **Typen** wird ???

HoverService

Protokoll in Abschnitt Hover-Effekt

Der HoverService lässt die Entwickler*innen Zusammenfassungen für jedes Element aufrufen. Um zu bestimmen, welche Zusammenfasssung zu welchen Element übermittelt werden muss, wird zu Beginn jeder Hover-Anfrage die kleinsten Nodes verschiedener Typen bestimmt.

Zusammenfassung	AST-Node-Typen
Variable	arg_block, ref_block, multi_block
EntityIdentifier	identifier_statement
Konstante	enum_constant
Kommentar	comment_block
PackageElement	package_block, struct_block, enum_block, entity_block, interface_block
Import	import_statement
Model Deklaration	model_declaration
DMF Deklaration	dmf_declaration

Das Generieren der Zusammenfassungen folgt für jede Zusammenfassung dem gleichen Ablauf. Zuerst werden anhand der AST-Nodes die Elemente aus dem semantischen Modell bestimmt. Aus diesen Elementen wird eine spezifische Datenstruktur erstellt. Mithilfe dieser Datenstruktur kann das jeweilige Template ausgeführt werden.

Nutzung von Golang-Templates

Die Standard Bibliothek von Golang bietet eine Template-Engine das Generieren von Texten genutzt wird. Die API besteht dabei aus 3 Komponenten: Templates, Funktionen und Datenstrukturen.

Listing 11: Ablauf der Template-API

```
1  // Inkludierung der Templates
2  //go:embed template/*
3  var tmplFiles embed.FS
4
5  // Vorbereiten der Templates
6  templates = template.Must(template.New("").
7              Funcs(funcMap).
8              ParseFS(tmplFiles, "template/*"))
9
10 // Generieren mit Templates
11 buffer := bytes.NewBuffer(make([]byte, 0))
12 err := templates.ExecuteTemplate(buffer, "variable", data)
13 return buffer.String()
```

Die Templates werden in eigenen Dateien abgelegt. Mithilfe der Golang-Embeded-API werden diese Dateien in die ausführbare Datei kopiert und können während der Ausführung ausgelesen werden.
Ein Template wird immer mit einer Datenstruktur aufgerufen. Innerhalb eines Templates werden Kontrollstrukturen und Zugriffe auf die Datenstruktur innerhalb von Statements mit '{{' und '}}' gekennzeichnet. Statements in Templates folgen einer simplifierzierten Golang Syntax. Das Ergebnis nach ihrer Evaluirung wird in das Ergebnis eingefügt. Die größte Simplifizierung sind die Blöcke der Kontrollstrukturen. Die Beginnen immer mit einem Statement in der Syntax des jeweiligen Keywords und enden mit dem '{{end}}'-Statement.
Ein Template wird definiert mit dem Define-Statement ('{{define "name"}}'). Der Name im Statement wird später zum Aufrufen des Templates genutzt.
Die Übergebene Datenstruktur wird immer mit '.' adressiert. So kann eine Variable der Datenstruktur mithilfe von '{{.Variablenname}}' eingefügt werden. Es ist auch möglich lokale Variablen zu definieren, z.B. '{{$importSlice := getImports .}}'. Ein Name einer lokalen Variable beginnt immer mit dem Zeichen $.
Funktionen werden ohne Klammern aufgerufen, jedoch können Klammern zur Abgrenzung der Parameter verschiedener Funktionen genutzt werden. Das Statement 'ne (len $importSlice) 0' wird zu einem Wahrheitswert evaluieren. Der Aufruf der Funktion 'len' befindet sich in Klammern sodass das Ergebnis als ein Parameter der Funktion 'ne' verwendet wird.
Mithilfe von 'if' und 'range' Statements kann der Ablauf eines Templates gesteuert werden. Ein 'if'-Statement funktioniert als Kontrollstruktur die einen Teil des Templates abhängig von einer Bedingung ausführt. Es kann mit einem 'else'-Statement erweitert werden. Ein 'range'-Statement übernimmt die Funktion der Schleifen. Es wird immer durch eine Slice iteriert, z.B. '{{range $index, $use := .Uses}}'.
Es können andere Templates aufgerufen werden. Dabei kann nur ein Parameter(=eine Datenstruktur) übergeben werden, weshalb häufig Methoden mehrere Parameter in eine Datenstruktur kombinieren. Ein Aufruf Statement enthält neben dem Parameter auch den Namen des Templates: '{{template "parameter"$element}}'.
Die Funktionen werden in Go geschrieben. Eine Map dient zum Aufruf der Funktionen während der Generationen. Zwischen Funktionen, Templates und Datenstrukturen wird während der Kompilierung und der Vorbereitung der Templates keine Typ-Überprüfung durchgeführt. Besitzen die Parameter während der Generation die falschen Typen, wird die Generation mit einem Fehler abgebrochen.

Funktionen besitzen keine Referenzen zum Kontext indem die Generation gestartet wurde. Deshalb bleiben nur noch die globalen Elemente und die Parameter. Globale Elemente können jedoch diesen Kontext auch nicht speichern, wenn die Generation parallel ausgeführt wird. Somit wird jeder erfordlicher Kontext in den Parametern übergeben.
Die Datenstrukturen werden normal im Go-Code definiert. Sie enthalten neben den Daten aus dem semantischen Modell auch den evtuellen Kontext für den Aufruf für eine Funktion oder bündeln Parameter für den Aufruf eines Templates.
Bei der Generation wird ein Writer, der Name des Templates und die Datenstruktur übergeben. Das Writer-Interface bietet eine Abstraktion, um Ziel und Implementierungs unabhängig Schreibvorgänge abbilden zu können. Um die Zusammenfassung später als String übergeben zu können, wird ein Buffer genutzt.

Die verschiedenen Zusammenfassungen werden im Abschnitt **??** beschrieben.

CancelService

Protokoll in Abschnitt Anfragen stornieren

Der CancelService wird zum Abbruch von nicht mehr benötigten anfragen genutzt. Um eine Abstraktion über alle Services zu ermöglichen, wird der Abbruch direkt in der Abstraktion der Verbindung implementiert. Innerhalb der Verbindung wird gepsichert, ob eine Antwort für eine Anfrage versendet werden soll. Wurde die Antwort vom CancelService blockiert, so wird die Antwort ignoriert. Um zu identifizieren, welche Antworten blockiert wurden, wird die ID der Anfrage genutzt. Diese wird auch in den Parametern der Abbruch-Anfrage übergeben.